# Architecture-Independent Distributed Query Processing

Hannes Fabian Mühleisen


Dissertation submitted to the

Department of Mathematics and Computer Science

Freie Universität Berlin

MMXII

Advisor: Prof. Dr.-Ing. Robert Tolksdorf

Second Advisor: Prof. Dr. Wolfgang Nejdl

Defended: December 7th, 2012

# Contents

# 1 Introduction

In recent years, the volume of data that is stored and processed has outgrown even wild expectations. To handle these amounts of data, immense computing and storage capacities are required. The long and successful history of supercomputers that can provide this level of performance has shown that such systems can indeed be built and maintained. However, the actual installations of these systems have been scarce due to the high costs connected with their operation. In the meantime, scale effects, technological advances and fierce competition have driven down these costs for smaller, "Desktop"-class computers. For a fraction of the cost of a supercomputer one can instead operate a large amount of these smaller computers, which – in sum – provide the same computing and storage capacities.

But therein lies a problem: A supercomputer uses sophisticated, fast and reliable technologies to internally distribute computation and storage, whereas large amounts of smaller systems operate fully independently, and are only equipped with slow, unreliable communication media. To beat the supercomputer, a plethora of mechanisms to co-ordinate tasks between them have been proposed and successfully implemented. Distributed systems based on cost-effective hardware were successful in driving down costs for large-scale computing needs. For example, they are a crucial precondition for maintaining a search index of the entire web solely financed by advertisement.

Being able to store ever-increasing amounts of data is a crucial precondition for many scientific and industrial applications. Building distributed storage systems out of smaller machines using coordination mechanisms was therefore a obvious development. However, it is impossible to create one single coordination mechanism for all challenges in distributed storage. Of course, we expect a system to excel in a multitude of dimensions at the same time. In the case of distributed storage systems main goals include the possibility to be expanded

without much hassle, to be reliable in operation, to provide high performance in data access, and the ability to adapt to new situations. Unfortunately, it has been conclusively shown that these goals compete among themselves, and any specific approach can only represent a trade-off between them. Starting from the requirements of individual organization, a large number of distinct mechanisms have appeared. These mechanisms range from being managed on a central location to "democratic" or fully distributed models, where each participating node takes part in the creation of a coordination mechanism.

Being able to store data in a storage system is not useful without a way of accessing the stored data. As applications became more complex, so did the access methods to their data. Retrieving data from such a system by – for example – a file name was acceptable in the past. Today, this concept grew more and more out of proportion with expectations, especially as system designers tried to incorporate more and more concepts from the database world, with its deeply structured data model and complex query models. However, while viable solutions to provide these concepts exist for single machines, distributing them has proved to be difficult to say the least. Therefore, the support for complex queries has deliberately been omitted, especially in fully distributed storage systems.

## Motivation and Research Problems

Support for fine-grained access to data held in distributed storage through complex queries is not a nice-to-have feature, but a necessity. Therefore, further research on the nature of distributed storage and on the possibilities for complex queries is required. However, previous and ongoing research is often tied to specific coordination methods. But therein lies a problem, as an infinite number of coordination models is conceivable, and no year passes without a new approach being proposed. Unfortunately, due to the close ties of previous proposals for complex query support with their coordination mechanism, they cannot be readily applied to the new model. Predicting their feasibility and efficiency of support for complex queries is difficult or even impossible for new models.

The main goal of this thesis is to separate query processing from the coordination model. We are trying to find a solution that intentionally ignores all the particularities of the specific

coordination model. This way, we can research distributed query processing from an abstract viewpoint, enabling us also to make observations and develop methods that are applicable to many models and systems. However, achieving effectiveness is not the only goal. In a connected network, all data can be found by flooding a request to every node. Unfortunately, the number of messages required here increases at least linear to the network size. In the literature, a logarithmic relationship between required messages and network size is considered to be desirable. Therefore, we will also investigate what the minimal requirements for this behavior are in our problem space.

The first research problem for this thesis is therefore to abstract from a specific mechanism and instead define a high-level model and coordination mechanism for distributed storage systems. This model is aimed to encompass the main classes of coordination mechanisms previously proposed. The research question here is if we are able to provide an abstract model that contains a good compromise between simplicity and complexity and whether this model exhibits sufficient performance to support higher-level algorithms.

Based on this model, the second research problem then is to develop a baseline method to provide a fine-grained access through complex queries to the stored data. The main questions here are whether it is possible at all to provide complex query processing on all data in an abstract distributed system. If so, the question of whether this method is efficient enough to be applicable is raised immediately. A major goal here is therefore to describe a notion of efficiency not in an absolute way, but dependent on the properties of the environment, which our underlying network model abstracts.

## Contributions of the Thesis

Aiming at finding solutions for our research problems, we contribute an abstract network and coordination model for distributed systems in general. This model is based on probabilistic behavior. We show how this model can be implemented by representatives of the main classes of distributed systems architectures, thereby confirming its flexibility. We also develop a structured data and query model that does not rely on central schema knowledge, thereby being perfectly suited for distributed environments. Based on these models, our

main contribution is a method for distributed complex query processing as well as a set of abstract requirements for efficiency in fully distributed query processing. By being based on our abstractions, this concept is potentially applicable to all environments that can be described by our models. Using this method, we are then able to also contribute experimental results that show the impact of environmental properties to the efficiency of distributed query processing. Also, we show how logarithmic efficiency with regards to our cost model and the number of nodes in a system can be achieved by using only our limited set of preconditions, namely probabilistic routing and a data placement strategy showing key locality.

## Scope and Methodology

Since distributed query processing on an abstract probabilistic coordination model has not been studied in great detail yet, we focus on the most immediate questions in this thesis. Our main areas of work are the correct evaluation of complex queries, without any central governing authority. In this context we describe the basic algorithm for query evaluation. Furthermore, to improve efficiency, we also discuss cost-based query optimization within our environment.

The methods we use to work on our research problems are spread out wide: We start by performing a literature analysis of previous work in distributed systems, distributed storage, and distributed complex queries. Based on the results of this analysis, we use modeling techniques to describe our abstract network model. To predict the average-case performance of the algorithms presented in this work, we use stochastic analysis. Finally, we perform a number of controlled experiments in a simulation environment to verify our predictions.

## Literature Connections

The network model and routing method presented in this thesis is based on previous research on a distributed storage system for RDF data based on swarm intelligence. The relevant publications are:

- H. Mühleisen, T. Walther, and R. Tolksdorf. A survey on self-organized semantic storage. *International Journal of Web Information Systems*, 7(3):205–222, 2011a

- H. Mühleisen, A. Augustin, T. Walther, M. Harasic, K. Teymourian, and R. Tolksdorf. A self-organized semantic storage service. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services (iiWAS2010)*, 2010

- H. Mühleisen, T. Walther, and R. Tolksdorf. Multi-level indexing in a distributed self-organized storage system. In *IEEE Congress on Evolutionary Computation*, pages 989–994. IEEE, 2011b. ISBN 978-1-4244-7834-7

- H. Mühleisen, T. Walther, and R. Tolksdorf. Data location optimization for a self-organized distributed storage system. In *Proceedings of the Third World Congress on Nature and Biologically Inspired Computing*, NaBIC '11, pages 176–182. IEEE, 2011c. ISBN 978-1-4577-1122-0

- H. Mühleisen and K. Dentler. Large-scale storage and reasoning for semantic data using swarms. *Computational Intelligence Magazine, IEEE*, 7(2):32 –44, may 2012. ISSN 1556-603X

The query processing method presented in this thesis is based on these publications:

- D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000 (Generic Query Processing Architecture)

- V. Papadimos and D. Maier. Mutant query plans. *Information and Software Technology*, 44(4):197–206, 2002 (Mutant Query Plans)

- R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4 (Moving Query Plans)

- H. Mühleisen. Query processing in a self-organized storage system. In *Proceedings of the VLDB PhD Workshop, co-located with 37th Intl. Conference on Very Large Databases (VLDB2011)*, 2011

## Organization of this Thesis

In this thesis, we start with previous work on distributed storage in Chapter 2. This non-original chapter gives an overview over the competing goals and the different classes of coordination mechanisms that represent specific balances between those trade-offs. From a literature review, we identify scalability, consistency and availability as the main competing goals for distributed storage, and also show how previous architectures make their compromises between them. Furthermore, we present the different data models and access methods from file systems to relational databases. We show that fully distributed systems with granular data access comparable to relational databases are highly desirable. Also, we describe the state of the art in distributed query processing.

Using the lessons learned form our literature review, Chapter 3 describes the models we develop in an effort to remove the effect of specific architectures or coordination models. We present a model for network structure based on the general concept of random networks, and also our main concept of probabilistic routing, which does not require the coordination model to produce exact results. Furthermore, we present an abstract algorithm for single data items. Starting from the relational data model, we also introduce a schema-less data model. Furthermore, we discuss the issue of data placement inside a distributed storage system, and show how locality in data placement is a central precondition for retrieval efficiency. Finally, we present our model for complex queries, which is also based on the relational model, in particular the both concise and complex class of Selection-Projection-Join queries.

Having laid the foundation for distributed complex queries, Chapter 4 presents our approach for distributed complex query processing, named “Mutable Moving Query Plans”. Here, the query evaluation process is continuously re-optimized while being on a journey through the distributed system. This allows the exploitation of local information as it becomes available, and eliminates the need for centralized knowledge. We present an abstract

algorithmic description and a cost model based on the notion of shipping cost regarding the total number of transmitted tuples. We also discuss methods to recover from failures, and present an abstraction and stochastic analysis of the average efficiency that can be expected from our proposed method.

In Chapter 5, we outline our rationale towards the use of controlled experiments to verify our predictions from the previous chapters. We also describe our simulation test environment, where we have fully implemented our proposed algorithms, while maintaining independence of specific network architectures. Using the industry-standard TPC-H benchmark, we then test the impact of environment parameters such as the size of the network to the cost of query processing.

Finally, Chapter 6 concludes this thesis, synthesizes the conclusions of the previous chapters, and lists the areas that merit future work.

# 2 Distributed Storage – Architectures and Interfaces

The process of organizing the storage and retrieval of data on more than one computer is a hard problem, and as with many computer science problems a single ideal solution is not at hand. This comes to no surprise, since the use cases for specific solutions are very diverse, ranging from reliable data archival to locating the newest Rolling Stones album on file sharing networks. Thus, as requirements for solutions diverge, so do the criteria to measure their performance. However, the design of a single solution always represents a conscious trade-off between different competing goals.

In this chapter, we first define our central concept of a distributed storage system and identify the generic goals scalability, consistency and availability for DSS from the literature. Then, we describe the three main classes of distributed architectures ranging from centralized systems to structured and unstructured Peer-to-Peer models. For each architecture, we describe its conceptual ability to meet the generic goals and give an example of an implementing DSS. We then continue to introduce the main data and access models for DSS, namely file systems, relational databases, Key/Value stores and RDF storage systems. By doing so, we are trying to show two main aspects: First, no architecture is able to reach all goals simultaneously; its selection is dependent on the compromises the system's users are willing to make. Second, selective access models such as relational database queries are an important precondition for efficiency, and supporting such access models over the whole range of system architectures is desirable. We then continue to show the state of the art in distributed query processing.

A Distributed Storage System (DSS) is a special case of a distributed system with the sole purpose of managing data, which has been defined by Tanenbaum as “A distributed system is a collection of independent computers that appears to its users as a single coherent system" [Tanenbaum and van Steen, 2002]. This view is twofold, it both describes the systems exterior view as a single coherent system as well as describing what the system consists of, namely a collection of independently operating computers. This is known as the “single system image”, and a major design goal for this class of systems. The definition also holds for distributed storage systems, they offer storage services to applications that are ignorant of the complexities inherent in distributed storage. A DSS is also made of a collection of independent computers, commonly called “nodes”. It should be noted that “independent” does not imply organizational independence, but only independent logical computers. Hence, nodes follow the “Shared-Nothing” paradigm [Stonebraker, 1986; Özsu and Valduriez, 1991]. We therefore define a DSS as being a *distributed system designed for the purpose of storing and retrieving data*.

Other definitions in the literature are compatible with this view, for example defining a distributed system as “A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively” [Singhal and Shivaratri, 1994]. This definition implies that a communication infrastructure is present between the nodes; otherwise the coherent external view would be impossible. Furthermore, a mechanism coordinating the nodes is in place. It is precisely the mechanism to achieve this coherent external view on a collection of nodes that will define the characteristics of a specific solution.

## 2.1 Goals and Dimensions

The motivations for deploying distributed systems are diverse, but by abstracting from specific requirements, researchers have identified sets of general design goals. In this section, we describe these abstract goals and motivations.

For distributed systems in general, Tanenbaum lists *Accessibility*, *Distribution Transparency*, *Openness* and *Scalability* as main goals [Tanenbaum and van Steen, 2002]. Accessibility refers to the sharing of resources between the involved parties; Distribution Transparency is the transparency of the physical location of resources, Openness to the interoperability of the system over implementation boundaries, and Scalability, which refers to the capability of the system to grow.

Distribution Transparency is subdivided into specific types of transparency. Access Transparency hides the physical access and representation of resources. Location Transparency hides the physical location (computer) of a resource in the system. Migration and Relocation Transparency allows resources to be moved to another location in the system, even while they might be in use. Replication Transparency allows several copies of a resource to be present in the system outside the knowledge of the system's users.

A similar set of motivations for distributed systems is constructed by Coulouris, *Heterogeneity*, *Openness*, *Security*, *Scalability*, *Failure Handling*, *Concurrency* and *Transparency* are listed [Coulouris et al., 2002]. Here, Heterogeneity is defined as the construction of a distributed dystem from different networks, operating systems, hardware components and programming languages. Openness is the abstraction between component interfaces and implementations. Security protects sensitive information when transmitted over the network. Scalability requires constant abstract cost in additional resources for supporting an additional user of the system. Failure Handling refers to the various possibilities for failure in the distributed system's components and the methods to deal with them. Concurrency allows concurrent access to the resources inside the system. Transparency hides the complexities of the system from application programmers.

Again, Transparency is divided into subtypes. In addition to the types of transparency presented by Tanenbaum, Concurrency Transparency allows several processes to operate concurrently on resources without interference, while Failure Transparency conceals faults in the system from applications. Performance Transparency allows the system to be reconfigured according to the current usage and load patterns, and Scaling Transparency enables the systems to expand without change to the system structure and applications.

Another set of goals is presented by Kshemkalyani, here *Distributed Computation*, *Resource Sharing*, *Remote Access*, *Reliability*, *Performance/Cost Ratio*, *Scalability* and *Modularity* are listed as main goals for distributed systems [Kshemkalyani and Singhal, 2008]. Distributed Computation refers to the need for a consensus between geographically distributed systems, Resource Sharing and Remote Access acknowledge the high costs to make all resources available at all sites as well as potential bottleneck issues at single sites. Reliability describes the inherent potential of increased reliability through resource replication and geographical distribution. Availability, Integrity and Fault-Tolerance are listed as sub-goals for reliability. The Performance/Cost ratio of a distributed system compared to specialized parallel machines is also listed as a possible motivation. Scalability is defined as avoiding bottlenecks as additional hardware is added through the system. Modularity extends this notion of Scalability, with hardware additions not hindering performance.

In the more specific case of DSS, these goals remain of course valid. However, the data-centric approach here leads to a shift in the emphasis put on these goals. For early distributed databases, *Transparency*, *Reliability*, *Performance/Cost Ratio* and *Scalability* were key goals [Özsu and Valduriez, 1991]. Transparency (with regard to Distribution and Replication) provided the single-system image mentioned above. Reliability was achieved through transaction protocols adhering to the Atomicity, Consistency, Isolation and Durability (ACID) principle [Haerder and Reuter, 1983]. Goals such as Accessibility and Concurrency are implicitly present, but less emphasis was put on Openness, Heterogeneity, Distributed Computation and Modularity.

However, even these reduced goals were shown to be incompatible. The Consistency-Availability-Partition Tolerance (CAP) theorem describes this dilemma for any DSS [Brewer, 2000; Gilbert and Lynch, 2002]. Here, Consistency refers to the consistent external view on the stored data according to the ACID principle, Availability defines the system's ability to generate an answer to each request, and Partition Tolerance allows the system to operate despite internal failures. It should be noted that Transparency is assumed here, and that Partition Tolerance is comparable to Reliability. The CAP theorem states that at most two of these goals can be achieved for any shared-data system. Fig. 2.1 shows the space created by these dimensions. The system S3 exhibiting Consistency and Availability is unable to

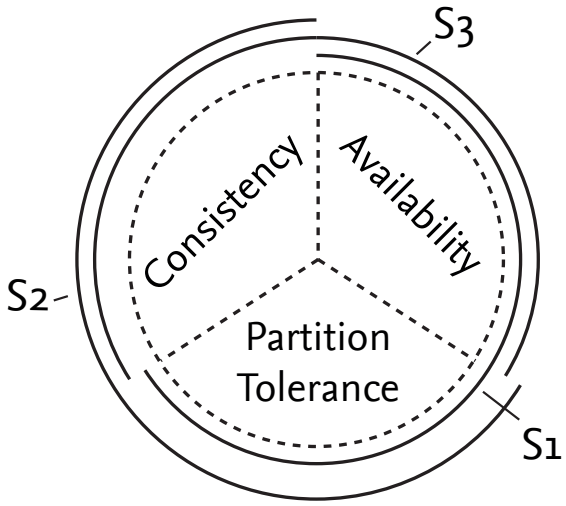


Figure 2.1: CAP Theorem – Dimensions [Brewer, 2000]

provide Partition Tolerance, as it is forced to reject requests in the event of a communication failure between the nodes. A well-known example for this class is a distributed database using for example the two-phase commit (2PC) protocol.

An even more concise description of the competing goals for DSS can be found in [Vogels, 2008]. Here, *Consistency* and *Availability* are presented to be the main competing goals. Furthermore, a specific trade-off between the goals is introduces with the notion of *Eventual Consistency*, which relaxes the consistency requirements in favor of availability.

In order to assess specific solutions for DSS, we now select a unified and small set of goals from the presented literature. These three competing goals are then used to assess specific solutions in the remainder of this work. The following sections discuss these goals and justify their selection. Accessibility, Transparency and Concurrency are implicitly assumed to be present, since this was also the case in the literature on the abstract goals for DSS. By the same token, we omit Openness, Heterogeneity, Distributed Computation and Modularity and Security.

### 2.1.1 Scalability

Arguably the most-often quoted reason for research on DSSs is Scalability. This broad term needs clarification. [Hill, 1990] discusses the term Scalability for multiprocessor computing, ultimately discouraging its use due to the lack of a rigorous definition. Merriam-Webster defines scalable as "capable of being easily expanded or upgraded on demand". In order to "upgrade" a DSS, several components could theoretically be upgraded, e.g. the number of nodes, their individual storage or processing capacity or the capacity of the communication method between the nodes. However, in the context of distributed systems, the term is generally applied only to the number of nodes.

Concerning the notion of "demand", in a DSS the demand to upgrade the system can come from two sides. Nodes in the DSS have a limited storage capacity, the amount of data that can be stored in the system is typically limited, and cannot be extended and managed infinitely. If more data is to be stored, more nodes have to be added to accommodate this. This notion is supported by the presented literature in a more specific form, as Scalability for DSS was defined as the ability to support additional data or users without non-linear increases in costs [Coulouris et al., 2002]. Furthermore, assuming that all nodes have a limited processing capacity, and every request for a stored data item requires some processing, the amount of requests the DSS is able to handle is also limited. If more requests are to be served in a given time frame, more nodes have to be also added to the system.

One of the competing goals for DSSs is thus the ability to handle increases in overall storage load and requests per time unit through the addition of more nodes to the system "easily". One important factor here is that even if the addition of more nodes itself is unproblematic, the concepts employed in the system may still impose limits on its scalability. For example, if all nodes have to register themselves at one central location in order to become a part of the DSS, the capabilities of this central location will limit the overall system's ability to scale. Thus, any central location or central data structure will pose a potential threat to scalability. This viewpoint is also supported by the presented literature, which required the absence of bottlenecks in the network for Scalability [Kshemkalyani and Singhal, 2008].

### 2.1.2 Consistency

The notion of a consistent view on the data stored in a DSS was introduced from the more general goals of transparency. In the presented literature, transparency to the physical location of data inside the DSS, transparency to replication of data, transparency to concurrent access and transparency to failures in the system were not only listed as goals, but as requirements. The main reason behind these various types of transparency is added simplicity for applications using the DSS [Özsu and Valduriez, 1991], these transparencies provide the single-system external image mentioned above. If these transparencies are present, the system provides a consistent view on the stored data. We can therefore regard a DSS to be consistent if it provides transparency in the four mentioned dimensions.

However, the absolute consistency required by the ACID principles was shown to be a major hindrance for the evolution of DSS [Gilbert and Lynch, 2002]. Therefore, its absolute definition has been relaxed into several consistency classes [Vogels, 2008]. These classes are defined from the application or client viewpoint using several independent processes, which write and read data to and from a DSS. How and when the processes see updates made to data objects inside the system illustrates the different consistency classes. In all cases, we assume a process having performed an update to one specific data object. *Strong consistency* guarantees all subsequent access to the DSS by all processes to return the updated value. Transactional systems typically provide this level of consistency after transactions have been committed. Contrary, *weak consistency* does not give any guarantees that subsequent accesses by any process will return the updated values. *Eventual consistency* represents a trade-off between the two extremes. The DSS guarantees that if no new updates are made to the data object, eventually all processes will see the last updated values consistently. The time period between the update and the consistency is referred to as the *inconsistency window*.

### 2.1.3 Availability

Availability was defined as a system's ability to generate an answer to each request [Brewer, 2000]. It therefore also subsumes the other goals reliability and robustness. Availability in a DSS can be seen from two viewpoints: First, the reliability of the entire system, whether

the system is able to reliably provide its intended services in spite of adverse real-world influences such as hardware failures. Second, the availability of access to the stored data, also present in spite of adverse influences. It is clear, that the second point is only relevant if the first point holds, if the system is dysfunctional all together, access individual data items stored is also impossible.

The availability on the system level is influenced on the availability of its nodes, but – more importantly – on their logical failure dependencies. To continue the previous example, if a central location encounters a failure, the entire system is no longer able to perform its indented service. Hence, this systems reliability is directly dependent on the availability of the central component.

If global availability is not the issue, availability of access to individual data items becomes an issue. As it is typically the case [Özsu and Valduriez, 1991], if the entire set of data is not stored on all nodes, the reliability of access is directly dependent on the consistency and availability of the single node or the set of replicated nodes for this particular data item. The method to achieve this consistency between nodes storing copies thus greatly influences availability of access to individual items.

### 2.1.4 Performance / Cost Ratio

The deployment of a DSS can also have a non-technical goals. The cost of creating a distributed single-image system out of a number of commodity off-the-shelf hardware (COTS) components instead of custom-building hardware with similar performance characteristics is often highly in favor of the distributed system. For example, the advertisement-financed web index run by Google is arguably only made possible through their use of commodity hardware [Barroso et al., 2003]. Another example is the creation of a supercomputer through the combination of over 1,500 game consoles at a fraction of the cost of a "real" supercomputer [Wood, 2011].

From an economic perspective, the differences in price between a large number of commodity components and larger single systems with comparable performance can be explained through the *economy of scale*. According to economics research, this difference could be explained through quick amortization of so-called set-up costs. As soon as the production

has been configured for the first “copy" of the product, the cost of producing the following products will asymptotically be determined only by direct costs such as labor or materials [Silvestre, 1987].

Hence, high-volume commodity hardware will in sum provide more power for the money than low-volume specialized hardware. Distributed systems can make use of this correlation in providing a higher performance/cost ratio, provided they are able to keep the synchronization costs for creating the aforementioned single-system image within strict limits, for example logarithmic to the number of nodes employed or even constant.

## 2.2 System Architectures

In the previous section, we have discussed the abstract goals for DSS. However, the means by which the performance along each dimension is to be achieved were not mentioned. In this section, we switch from this external view to an internal view, and review three distinct system architectures common in the construction of DSS: Centralized as well as structured and unstructured Peer-to-Peer (P2P) architectures. This taxonomy of distributed system architectures was also taken from the literature [Tanenbaum and van Steen, 2002].

Previously in this chapter we have identified the single system image as one of the main implicit goals of a distributed system, which is also present in the case of a DSS. Applications using the storage system are generally not interested in the specific location of a data item. Rather, they are keen on using the DSS the same way as a local storage medium. To achieve this level of consistency, a mechanism to coordinate the nodes is required. This mechanism has a high influence of the systems performance, determining its operational boundaries and its effectiveness in a particular environment [Placek and Buyya, 2006].

For the goals outlined above, each of the architectures represents a unique trade-off between them. In the following, we will point out where exactly the particular trade-off is taken, and how this impacts the performance for each dimension. We also give examples for systems implementing the coordination mechanisms or architectures. Since they are instances of the abstract architectures, restrictions identified for the architectures also apply to the implementations

### 2.2.1 Centralized Architectures

Centralized or clustered systems are the first intuitive approach in creating a single system image for a distributed system. One of the nodes is designated to be a “master” or “control” node. This node serves as a front-end to client applications and controls the flow of data within the distributed system. For example, the master node in a centralized distributed file system could keep a look-up table mapping file names to node addresses. All requests are sent to this master node, which can find out the actual node storing the file, and reroute the request accordingly.

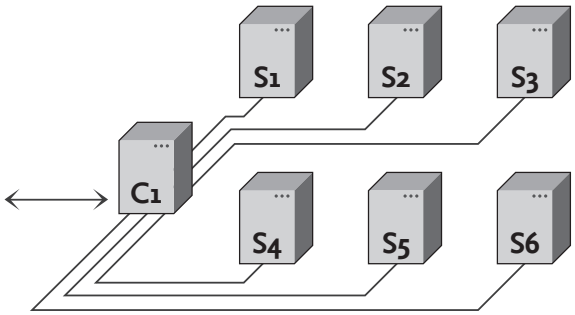


Figure 2.2: Network Structure for Centralized Storage

An abstract network structure for a centralized distributed system is depicted in Fig. 2.2: The control node C1 is accessed from outside, and maintains a single connection to the storage nodes S1 through S6. No connections have to be present between the storage nodes. In this setup, storage nodes do not have to be able to execute complex algorithms, they are little more than remote hard disks.

The centralized approach has several advantages: The amount of nodes in this system is only limited by the ability of the master node to organize them. Typically, organization of request processing is less expensive than the actual processing itself. Hence, a centralized system is able to scale to some degree, limited by the capabilities of the master node. Consistency is also high, since the master node sees every request, and does not need to communicate with other nodes to reach a consistent state in the system. However, the main disadvantage is availability: If the master node should fail or become overloaded, the entire

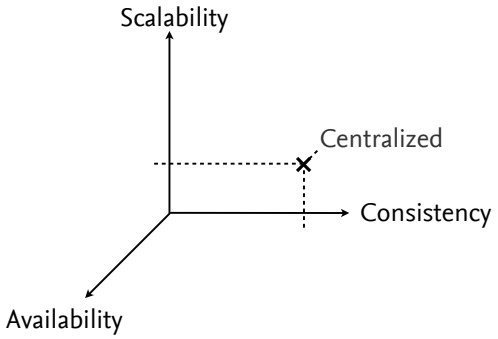


Figure 2.3: Centralized Architecture - Trade-Off

system is no longer able to function correctly, even if all the other nodes continue to function normally. Also, techniques such as sampling the requests to make the system adaptive are inexpensive when performed on the master node, as all requests are at least redirected there. The master node is then able to re-organize its storage nodes according to the current request patterns. This trade-off situation is depicted in Fig. 2.3

A well-known example for a centralized DSS is the Google File System (GFS) [Ghemawat et al., 2003]. The authors specifically cite economic reasons for the creation of GFS, as mentioned before in Section 2.1.4. A GFS installation consists of a single master server and multiple data nodes, here called “chunk servers”. Files are divided into chunks of a fixed size and stored on the chunk servers. Each chunk is assigned an identifier, and the master node contains a mapping from file names to chunk identifiers, and from chunk identifiers to machine identifiers. Client operations on the chunks are performed directly on the chunk servers for performance reasons.

From the viewpoint of scalability, a GFS cluster can easily be extended with new chunk servers, but scalability limited by the amount of operations that the master server is able to handle per time unit. The authors report some performance figures on this in their paper. They claim the master server is able to handle “thousands of operations” per second. In their evaluation they also report some performance figures on GFS. For example, in a cluster having about 500 MB/s traffic, the master node had to handle about 500 Operations per second. This represents a serious bottleneck close to the design limit for a imaginable 10-fold

increase in load. Due to this issue, GFS has recently been retired and replaced with a new multi-layered system "Colossus", where the file metadata previously stored on the master server was moved to its own DSS [Fikes, 2010].

For availability, the designers went to great lengths to secure GFS both against overall failure as well as against failure of access to single files. Using an arbitrary placement strategy, the master server stores each chunk on three different chunk servers for reliability. To protect against global failures, the state of the master server is also replicated, only committing to a change if all replicas have written the change to their persistent storage media. This enables starting a new master server, should the old one fail. Since the master server controls the replication, consistency can be ensured without communication overhead. Furthermore, if the master server has to be relocated to another machine, they rely on DNS changes to redirect client operations. While clearly being a center of attention for the designers, from a conceptual point of view the master server still threatens the systems reliability.

However, as a global viewpoint is available in the master server, it is able to adapt the DSS to current needs by re-balancing chunks between the chunk servers. For example, the disk space and load on the chunk servers are monitored and chunks are moved to a less-heavily used server if a chunk server starts to become overloaded. We can therefore observe a high adaptivity in GFS.

### 2.2.2 Structured P2P Architectures

Eager to remove single points of failure, researchers have devised the concept of a distributed system based on the P2P method. Here, all nodes have the same rights and responsibilities. They use their communication method to create the single-system image purely from bilateral interactions. In a structured network, all nodes adhere to a common global law, which governs their interactions. This law – in the case of implementing a DSS – defines the network structure created between nodes and which data items are being stored on which node. By honoring this global law, structured P2P systems generally achieve a high degree of efficiency.

The majority of Structured P2P systems use a variant of a so-called Distributed Hash Table (DHT). One of the originally proposed DHT concepts was CHORD [Stoica et al., 2001].

Here, the nodes create a ring-shaped network structure, and use a hash function to assign storage identifiers to node addresses. Furthermore, by creating shortcuts in the ring structure, they are able to find data for a specific identifier in a logarithmic complexity with regard to the number of nodes in the system.

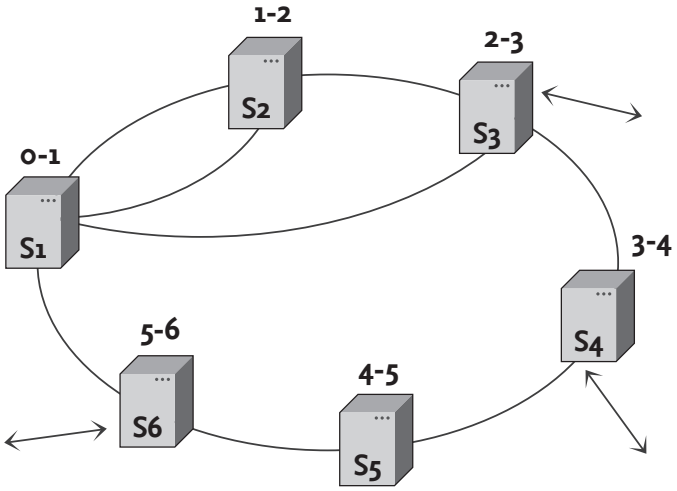


Figure 2.4: Structured P2P – DHT

Fig. 2.4 shows an example for a ring-shaped DHT: The nodes S1 to S6 create a ring-shaped network structure between them, and each node is assigned a range from the result range of a hash function, e.g. node S2 being responsible for all hash values in the range of $[1, 2[$. All data stored in the network is stored along with an identifier. This identifier is then hashed, which determines the node actually responsible for storing the data. An arbitrary replication strategy can secure data against node failures. Requests have also have to include that identifier, and regardless of which node receives a requests, it is able to route the request to the responsible node according to the global law.

A structured P2P network is able to deliver high performance combined with a high availability due to its equal rights approach combined with the efficient look-up procedure as defined by the global law. One node's failure does not affect the other nodes in their capability of performing their intended service. Scalability depends on the effort required to

add new nodes to the system, and can only be discussed for specific approaches. However, as we have seen, the absence of bottlenecks is also required for scalability. In structured P2P networks, load imbalances are an issue. Through distribution, they are also difficult to detect as compared to a centralized system and even more difficult to mend, as the global law cannot be changed by the system itself. Furthermore, it has been shown that the complexity of placing data in a structured P2P system such that request load is balanced is equivalent to $NP$ [Gribble et al., 2001].

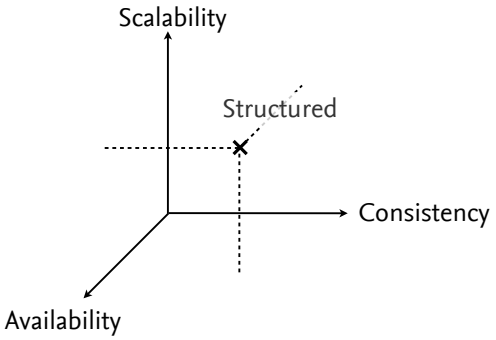


Figure 2.5: Structured P2P Architecture - Trade-Off

Consistency is generally also an issue, as every node shares the responsibility for the system. Since the node receiving a request and the node storing the data are seldom identical, ensuring consistency requires extensive communication in the network. Furthermore, replication schemes require updates at several locations in the network. Therefore, consistency requirements are often relaxed, as described in Section 2.1.2. The particular trade-off for structured P2P architectures is shown in Fig. 2.5.

An representative example for a DSS using a structured P2P as its coordination mechanism is “Dynamo” developed at Amazon [DeCandia et al., 2007]. It is a almost textbook example of the DHT approach introduced here. Its global law also defines a ring network structure and uses consistent hashing [Karger et al., 1997]. Consistent hashing has a crucial advantage for the systems scalability: With a conventional hash function such as SHA1, each nodes responsibilities for a specific data identifier range would have to be re-calculated for the entire network, resulting in the movement of a considerable portion of the data. By using

consistent hashing, this is not necessary, and the magnitude of the amount of identifiers to be remapped averages to the number of identifiers for one node.

By using the consistent hashing scheme, Dynamo is able to scale fairly well, as the addition of new nodes does not require effort on a high portion of the other nodes. A new node is assigned a random identifier range from the hash ring, and connects itself to the nodes adjacent to this range, which can be found in logarithmic complexity. As soon as the node is connected, the nodes previously responsible for these ranges transfer the data. However, in the hash ring scenario it is impossible to add nodes without affecting data placement, for example to reduce load. This can have a negative impact on scalability. Also, the authors themselves have identified the unfair data and request load distribution as an issue, with around 10% of nodes deviating over 15% from the average request load.

### 2.2.3 Unstructured P2P Architectures

A third approach is to retain the equal rights and responsibilities of the nodes in a distributed system, but to remove the global laws from the equation. Then, a heuristic of varying complexity can be used to forward requests to other nodes. An early example for such a system was the "Gnutella" system, where each node keeps a limited list of known nodes. Retrieval requests for data were flooded to all known nodes, and limited by a Time-to-Live (TTL) [Ripeanu et al., 2002]. The general concept assumes all data being stored on the nodes independent of the retrieval protocol, thus no single-system image is provided for the addition of new data.

Nodes can join the network by merely connecting to any nodes that are already part of the network. They will receive retrieval requests from them, which makes the data they store available to others, and are able to send their own retrieval requests into the network, making this kind of network particularly suited for highly fluctuating environments [Chawathe et al., 2003]. As long as the maximum path length inside the network can be traversed inside the TTL, data that is available will also be found. Thus, the addition of nodes and thus the concept's scalability is high. The immediate downside is the general low performance of operations. Since every forwarding operation requires some time, this class of systems generally exhibits low retrieval performance. Furthermore, the flooding of requests leads

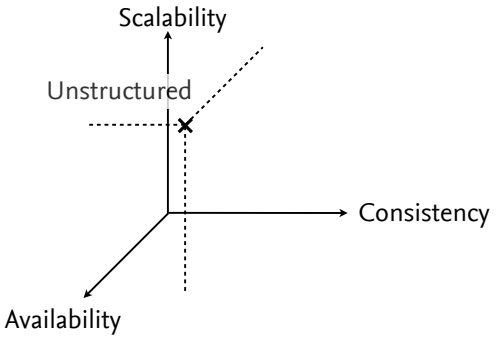


Figure 2.6: Unstructured P2P Architecture - Trade-Off

to a high amount of protocol overhead, sometimes exceeding 50% of overall traffic [Ilie et al., 2004]. Within the dimensions we have identified, the trade-off for unstructured P2P architectures is shown in Fig. 2.6.

Availability of the Gnutella-style systems is very high. If nodes fail to function, the entire system is not affected at all. In order to protect individual data items against failures, they could be replicated to other nodes. In contrast, the unfair load distribution identified for structured P2P systems is also present, as no synchronization or feedback mechanism is provided between nodes to react to overloaded nodes or network connections.

While the original Gnutella concept contained a routing heuristic that only flooded every request to every known nodes, further research has provided more complex and likely more efficient heuristics, aimed at increasing the performance as well as the adaptivity of the Gnutella concept [Lv et al., 2002]. The “Gia” concept for example proposed a search protocol based on biased random walk, that would only forward requests to non-overloaded neighbor nodes [Chawathe et al., 2003]. Combined with a topology adaption (also proposed by [He et al., 2008]) and flow control protocol they were able to increase the retrieval performance.

While the Gia approach is agnostic of the content that is requested in the network, our previous work on “S4” created a routing heuristic based on an ant-inspired positive feedback mechanism on the used identifiers [Mühleisen et al., 2010]. Here, successful operations were used to intensify virtual pheromone paths, which would influence future requests for the same or similar data identifier into the direction the previous request was successful. Also, a

write operation is offered on every node, making S4 a fully featured DSS based on heuristic routing. To keep the routing data structures on the nodes small, new data items are clustered together with similar data items. While being scalable, reliable and adaptive, S4 is unable to provide any guarantees as to whether a retrieval operation will find a data items, representing a unique trade-off.

### 2.2.4 Architecture Comparison

From our description of the different system architectures and the corresponding coordination models, we can now attempt a relative comparison of these approaches according to the goals we have identified. This comparison is pictured in Fig. 2.7.

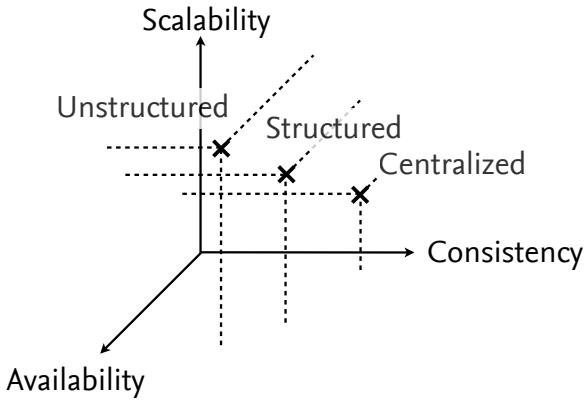


Figure 2.7: System Architectures and Goals

For centralized systems, scalability with regard to adding more nodes to the system is no issue, as they are registered at one central location. However, the ability of the system to handle more data or more requests is limited by the capabilities of the central node, which also represents a serious bottleneck. Hence, scalability is limited for centralized architectures. Consistency on the other hand is high through the central node. It has all the knowledge required to provide the different transparencies. Availability is difficult again, since the availability of the system is bound to the availability of the central node.

For structured P2P architectures, scalability to the number of nodes is high, albeit some reorganization of the network structure may be required. More data and more requests

can be handled through adding more nodes, but bottlenecks may arise through the global law controlling the distribution of data and requests. Hence, scalability is higher than in centralized architectures. Consistency is more difficult than in centralized systems, since costly communication between nodes is required. Contrary, availability is higher compared to centralized systems, as the damage through failures can be repaired, and replication can ensure the availability of data, even if the node responsible fails.

Unstructured P2P architectures exhibit the highest scalability when compared to the other architectures, since adding nodes does not require network organization. Bottlenecks and hot points can be repaired through local actions and are not bound to adhering the global law. Contrary, ensuring consistency is considerably lower than in the other two architectures, as no guarantees can be given of finding data inside the network. Availability of the entire system is higher than in the other two approaches, as failing nodes never affect the entire network.

## 2.3 Data Models and Access Methods

As much as DSS differ in the method they use to provide a single-system image to applications, as much they differ in the models they use to represent their data and the methods they offer to access the stored data. In this section, we present representative examples for common data models and access methods currently found in DSS implementations ranging from file system interfaces, key/value models to fully-fledged databases, both for relational and graph data. For each model, we give examples for each of the system architectures presented above.

### 2.3.1 File Systems

Historically, one of the first approach for DSS implementations was to provide a regular file system. The methods that have to be provided by a file system were most prominently defined as part of the Portable Operating System Interface (POSIX) standard. The data model defined by POSIX is an hierarchical directory tree. Files are assigned to a directory and described with some meta data such as size, access rights and ownership. File contents

are unspecified, and are read in byte-by-byte. POSIX consequently defines a set of system calls to manipulate the directory tree, file meta data, and to read and write file contents.

From a distributed perspective at file systems, development was at first focused on making a file system on a single server available to several client computers. The Network Filesystem (NFS) was the first of several methods to make this possible [Sandberg et al., 1988]. It implements the POSIX interfaces, and can thus directly be integrated into client computers file systems with a suitable driver, which allows applications to make use of NFS without their knowledge.

One step further in distributed file systems towards a DSS are centralized approaches such as XtreemFS, where a single master server holds the directory structure and file meta data, but where file contents are served from a number of storage nodes [Hupfeld et al., 2008]. XtreemFS fully supports the POSIX file system standards. The Google File System (as described in Section 2.2.1) follows a very similar approach, but only provides a subset of the POSIX operations due to performance concerns [Ghemawat et al., 2003]. This requires applications using GFS to also use a specialized GFS client program, removing file system transparency.

A fully distributed file system is "Ivy", where the file contents, directory structure and meta data are stored in the structured P2P system DHash [Muthitacharoen et al., 2002]. However, Ivy is also unable to provide conventional file system semantics due to concurrency issues. An evaluation has found Ivy's performance to be two to three times slower than NFS, which is still impressive considering its lack of central control.

While distributed file systems have clear advantages for simple applications such as logging or providing storage for a higher-level access method, their flat file data model prohibits access to very particular parts of the stored data.

### 2.3.2 Relational Databases

Relational databases are storage systems for structured data, which is logically available in the relational model. Using the standardized Structured Query Language (SQL) [Melton and Simon, 1993], applications are able to retrieve very precise subsets of the stored data from the relational model, thereby reducing communication costs and application complexity.

The relational model is based on first-order predicate logic, and defines the relationships between tuples and relations [Codd, 1970]. A *tuple* $t$ describes a single data entry (e.g. a single person) and is defined as a partial functions from a set of attributes to atomic values. A tuple's function can be given as a set mapping, for example $t := (\prime firstname\prime, \prime John\prime), (\prime lastname\prime, \prime Doe\prime)$. The domain of a tuple (the set of attribute values) is called its *header* $H$, in this case $H(a) = \prime firstname\prime, \prime lastname\prime$. A single relation $R$ is now formed from a single header $H_R$ and a set of tuples $B$, such that $t \in B \rightarrow H(t) = H_R$ holds. Furthermore, the relation is assigned an identifier $i$, formally $R = (B, H_R, i)$. A relational database typically contains a large number of relations. The union of all headers inside a database is referred to as the database *schema*, which is used to determine data partitioning and to verify query validity.

While relational databases are a very successful concept, databases quickly grew out of the capabilities of single computers. The distribution of data and the request load was identified as a viable solution [DeWitt and Gray, 1992]. Distributed databases typically use one or few processing nodes that have access to the database schema and are in charge of processing queries and handling updates to the database [Teradata, 1983]. While processing queries, processing nodes access data on a larger set of storage nodes. The processing nodes also determine the distribution of relations (either vertically by relation or horizontally by tuples) between the storage nodes. From a DSS perspective, this represents a centralized approach as described in Section 2.2.1. The distributed evaluation of complex queries is discussed in the following Section 2.4.

### 2.3.3 Key/Value Stores

While relational databases provided and still provide the back-end for a large range of applications, a need for a new class of storage systems was identified. The so-called "NoSQL" systems traded away complex queries and transactional guarantees against potentially higher scalability [Brewer, 2000]. While the class of NoSQL systems also includes other data models, the Key/Value Store (KVS) is among its more prominent examples [DeCandia et al., 2007]. The data model used by KVS can be compared to a set of tuples similar to those used in a relational database. However, contrary to relational model, every tuple in the system can

have a different header. Hence, no relations and schemata can be defined. It is precisely this lack of global structure that KVS draw their advantages in scalability from. Furthermore, no complex query language is defined, thereby moving the complexity of complex data retrieval processes into applications.

One of the first representatives of distributed KVS was "BigTable" from Google [Chang et al., 2006], which is built using GFS (see Section 2.2.1). Hence, BigTable is also a centralized DSS, but in contrast to GFS it provides storage for structured data. A BigTable is a multidimensional and sorted map. A row key and a column key index the map. A table may have an unbounded number of columns, but they are organized into "column families" to group columns for storage optimizations. BigTable has no declarative query model, instead, data is explicitly requested from the virtual map either using explicit row identifiers or a column specification.

As mentioned, KVS lack schema information and complex query models, and are thus inherently more suited for distribution. The KVS "Cassandra" developed at Facebook [Lakshman and Malik, 2010] provides a data model identical to BigTable, but – similar to the Dynamo system mentioned in Section 2.2.2 – provides a fully distributed implementation by also using a structured P2P network for coordination. As guaranteeing replication in a P2P system is no easy task, Cassandra allows applications to specify their desired consistency level on every insertion operation.

Even though KVS do not provide complex query facilities, some effort has been invested in making their interfaces more compatible with users familiar with relational databases and the SQL query language. For example, the "Google AppEngine datastore" interface provides a SQL-like interface to the background map [Chu-Carroll, 2011]:

```
SELECT * FROM Person
        WHERE last_name = 'Wowereit'
        AND birth_year >= '1953'
        ORDER BY last_name
    LIMIT 5
```

In this example, `Person` describes an BigTable map and the filtering properties `last_name` and `birth_year` are column names. Using an automatically created index on the column names, the KVS is able to retrieve all row names matching the selection criteria and then applies the `ORDER` and `LIMIT` operations afterwards. While certainly being a helpful shortcut for developers, the lack of cross-table operators such as the join operator makes the system come nowhere close to a relational database.

### 2.3.4 RDF Storage Systems

With the advent of the Web, researchers have sought to interconnect the data available there. Relational databases are difficult to connect over different schemata and over system and organizational boundaries. To overcome this issue, the Resource Description Format (RDF) data model has been created. An RDF graph is a directed graph, where the nodes are either URIs, literal values such as strings and integers, or graph-internal identifiers known as blank nodes. Directed and labeled arcs connect these nodes, URIs are also used for these labels. RDF provides a highly generic and flexible data model able to express many other more specific data structures, such as relational or object-oriented data. Furthermore, each participating organization is able to create their own schemata within the flexible graph model and also refer to resources on other systems using URLs. This makes large-scale data integration and also their consumption possible.

More formally, let $\mathcal{U}$ be a set of URIs, $\mathcal{L}$ a set of literals and $\mathcal{B}$ a set of graph-internal identifiers ("blank nodes"). Any element of the union of these sets $\mathcal{T} = \mathcal{U} \cup \mathcal{L} \cup \mathcal{B}$ is called a RDF *term*. A RDF *triple* is a triple $(s, p, o)$, where $s \in \mathcal{U} \cup \mathcal{B}$, $p \in \mathcal{U}$ and $o \in \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. An RDF *graph* is defined as a set of triples [Hayes, 2004]. This strict formal definition also allows making general assertions about resources and their relationships and binds them to a vocabulary. A "reasoning" computer program is then able to calculate the data only implicitly stated in the data.

The query language SPARQL has been developed to express complex queries on RDF graphs. For convenient access to the data stored in such a graph, one may use a so-called triple pattern, which may contain variables instead of values. Variables are members of the set $\mathcal{V}$, which is disjoint from $\mathcal{T}$. A *triple pattern* is a member of the set $(\mathcal{T} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$.

The declarative complex query language SPARQL is based on combining triple patterns and provides many additional features such as ordering and filtering [Prud'Hommeaux and Seaborne, 2008].

According to our recent survey [Mühleisen et al., 2011a], storing RDF graphs in a distributed way creates a number of format-specific challenges. These challenges make it difficult to design a storage scheme that can be shown to be efficient for all possible cases [Battré et al., 2006]. The usage of vocabularies along with practicability reasons typically creates a very small frequency of values for the predicate entries. The even smaller set of standardized properties for example for resource type definition further exacerbates this problem. Experiments have shown that over 90% of queries use one of these properties, inevitably creating hot points within the data set [Harris et al., 2009].

## 2.4 Distributed Query Processing – State of the Art

In the previous chapter, we have seen how the architecture and the coordination mechanism used in a DSS is one of the main factors determining the systems' performance in various dimensions. We have also seen how precision data access for example provided by the relational access model using declarative query languages is crucial for efficiency. In this section, we therefore review the state of the art in distributed query processing for centralized and structured as well as unstructured architectures. Our goal here is to show the tight integration of previous approaches with the respective architecture.

Distributed Query Processing (DQP) is the process of retrieving and aggregating data from multiple sources according to a declarative result description. Typically, processing starts with a abstract and declarative complex query, which is then translated and optimized into an precise executable description. The general goal of this optimization is to minimize cost in various dimensions; oftentimes the time to result is used. While research on DQP dates back several decades, only in recent years both the need and the infrastructure has arisen for its realization. For some time now, all major commercial database systems have supported DQP in some way. [Kossmann, 2000] presented a reference architecture for DQP in relational

databases. He claims that to present an approach on DQP that is suitable to implement DQP on *any* kind of distributed system.

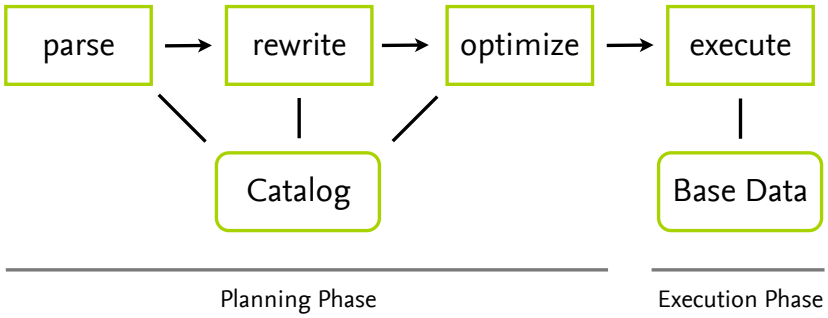


Figure 2.8: Query Processing – Generic Process

This reference architecture is shown in Fig. 2.8: In a first step, a query is *parsed* from a textual representation into an abstract internal representation [Pirahesh et al., 1992], very similar to the query model presented later in Section 3.5. Then, the *rewrite* component transforms the query tree by applying transformation rules, that are independent of the data stored. The query is then *optimized*, which makes use of the global schema information and statistics stored in the *Catalog*. To find the optimal way of executing the query (the *plan*, the optimizer enumerates alternatives and ranks them according to a cost model, which encompasses the optimization goals. This plan is then *executed* on the data stored in the DSS.

From the DSS point of view, the most critical component here is the catalog, which, as we will show in our discussion on the data model in Section 3.3, represents a global state which is a threat to scalability and consistency [Tanenbaum and van Steen, 2002]. Most previous approaches on DQP are based on the assumption of a centralized coordination method. Since we are focused on performing DQP on various types of DSS, we will present a number of approaches on distributed DQP on non-centralized coordination mechanisms in the following.

### 2.4.1 Structured P2P Query Processing

[Harren et al., 2002] present an approach on performing DQP inside a DSS using the structured P2P coordination approach. Interestingly, they also disregard the idea of a global schema; they rather focus on "natural attributes" in the data, very similar to the Key/Value approach. The main idea for query processing is to use the distributed hash table to store and route tuples to their destination. They have extended the DHT interface with a notification hook, which enables their query processing layer to react on tuples being added to the current node's storage. Join operations are now evaluated by taking all tuples from the joined relations and inserting these tuples into the DHT with the join attribute as key. As the tuples arrive at the node responsible for storing that key, their query processing component is notified and the result tuple can be created and passed on to the parent operator in the plan. However, as mentioned before, the authors observed a grave load distribution problem with this approach, which is only exaggerated by uneven data popularity or skewed key distributions.

[Triantafillou and Pitoura, 2003] attempt to create an unifying framework on DQP in networks also using a DHT. Tuples are inserted into the DHT using an order-preserving hash function. Additionally, each tuple is inserted multiple times, each time using a different Key/Value pair as the insertion key. Hence, they are able to support range queries, as all values in the range specification are stored on subsequent hash values. Furthermore, join processing is very straightforward: Tuples with identical values for a join attribute are will have replicas stored on the same node. While they support the relational data model, they do not present a method for schema distribution and also mention this issue a one of the major obstacles for the scalability of their approach.

[Brunkhorst et al., 2003] propose query processing in a two-layer P2P network: Each node periodically advertises the relational schema of the stored data at a limited number of "Super-Peers", which are assumed to have higher availability and performance than the storage nodes. The schema information is then replicated between the Super-Peers. Queries to be evaluated are sent to the Super-Peer, with the possibility of carrying executable code for operator evaluation. On the one hand, this greatly extends the expressivity of the queries,

as each user can provide its own operator implementations, on the other hand this creates an issue, as the costs for executing unknown operators are also unknown. More importantly, they show how static optimization is not possible in their environment and execute queries. The rather centralized query execution is performed by finding the nodes storing data relevant to the query from the aggregated schema information and generates partial queries for each node, with the results being collected at the node that started the query. Each operator in the query tree is annotated with the address of the node responsible for evaluating the operator. The cost model prefers queries with the smallest number of subqueries. However, since query optimization assigns each each operator a node, the location of data has to be known, which is impossible with heuristics-based coordination models. A very similar approach has also been proposed in [Kokkinidis and Christophides, 2004].

[Avnur and Hellerstein, 2000] recognize the need for *continuously adaptive query processing* in the DSS environment. They have proposed the "Eddy" concept, which is an abstract routing unit routing tuples to operators during query execution. Furthermore, an eddy is able to reorder operators on-the-fly, considerable reducing complexity from the initial optimization. They have identified "moments of symmetry" during query processing, where it is possible to change the order of execution for the operators to be evaluated. Furthermore, they acknowledge the lack of complete information, and introduce a notion of varying estimates of cost for operator costs, operator selectivity, and tuple flow rate from their sources. An Eddy is executed on a single node, which builds a fixed data flow graph ("River") for tuples from remote sites into eddy. Favoring adaptivity over best-case performance, their join algorithm allows changes to the query execution plan. Also, operator selectivity, which is crucial for accurately estimating costs, is "learned" during query processing with a lottery scheduling algorithm [Waldspurger and Weihl, 1995]. Thus, future queries enjoy more accurate optimization.

[Rösch et al., 2005] describe best-effort query processing on a DSS using the multi-dimensional Content-Addressable Network (CAN) [Ratnasamy et al., 2001]. Through the use of a clever hash function, where both relation id and the tuple identifier are hashed and the result bit-wise concatenated. The resulting bit string is split into coordinates, tuples are sored along a Z-Curve in the CAN content overlay. This concept is pictured in Fig. 2.9.

Similar to [Triantafillou and Pitoura, 2003], they also store every Key/Value pair from every tuple in their overlay, enabling retrieval by attribute values. To save space, these additional entries do not contain the entire tuple, but rather a pointer to the tuple identifier. However, this requires an additional look-up process. To retrieve data from their Z-Curve data structure, they introduce a retrieval operation aimed at a specific part of the curve. Join processing is performed consistent to [Harren et al., 2002], where the tuples to be joined are added to a temporary relation in the DHT.

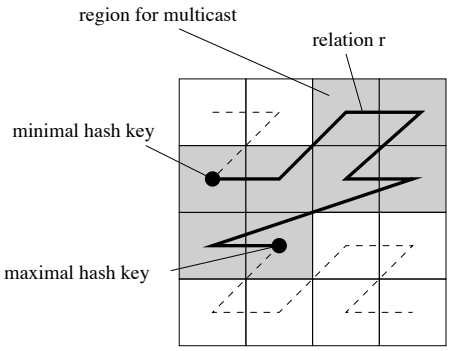


Figure 2.9: Z-Curve Addressing in CAN [Rösch et al., 2005]

For query processing, the Eddy concept from [Avnur and Hellerstein, 2000] is used. The node that received the query creates the Eddy processing node and the corresponding "River". The query tree is sent to all nodes potentially storing data relevant to the query as determined from the Z-Curve. Each node processes the plan and executes all local operators. If an operator cannot be processed locally due to missing data, the remaining plan and intermediate results are sent to the peer responsible for the next operation. If the root of the query has been marked as evaluated, the result of the query is sent back to the node where the query originated.

The authors identify an important question while performing DQP: From a given query tree, the most important decision is *which operator is to be evaluated next*. Hence, every query plan is extended with a "ToDo" list of the order in which the operators are to be evaluated. The strategies they present to populate this list are a random strategy, a priority strategy, and a strategy preferring the join operations first for which the distance to the home

peer is minimal. Their evaluation shows how the distributed Eddy outperforms a centralized approach, but also shows that the central Eddy collecting all results can become a bottleneck.

### 2.4.2 Unstructured P2P Query Processing

The method for DQP presented by [Karnstedt et al., 2004] was first to not require a structured P2P coordination mechanism in the DSS. Rather, their approach is based on the concept of a *routing index* [Crespo and Garcia-Molina, 2002]. From their XML data model and complex query language, each node stores the amount of elements it can retrieve from each neighbor node for a subquery (XPath expression). This information is disseminated through the network graph, but with a limited distance. Each node aggregates the routing index data received from the neighbor. Through this aggregation, the routing index stays scalable, as the amount of data inside the index is limited. At the same time, through multiple hops in the network, the data remains accessible. However, this also leads to the degradation of their routing index to become an error-prone heuristic. Nodes cannot determine the location of data for data stored too many hops away through the network. Hence, their coordination model closely resembles the "biased random walk" presented in Section 2.2.3.

For query processing, they recognize this limitation of their routing index and thus keep query plans adaptable. As they are routed through the network, they can be adapted with the data found in the routing indices of the nodes on the way. Their evaluation compared the routing methods flooding, a complete routing index with global knowledge, and their "limited horizon" index. They compared these methods according to the number of messages between the nodes and concluded on the need for query shipping in this environment. However, by using a content-based routing index, they have not considered network-centric heuristics such as the one presented by [He et al., 2008].

[Dedzoe et al., 2010] present an approach to evaluate Top-K queries on unstructured distributed systems. In Top-K queries, only the K "best" results are returned according to a relevance score [Ilyas et al., 2008]. They argue that since in an unstructured P2P network, total performance of complex query processing would otherwise be limited by the performance of the slowest node. By allowing a variable notion of response quality, they are able to remove slow nodes from the evaluation process. However, their method of finding

nodes inside the network is flooding, thus making their approach less scalable as discussed in Section 2.2.3.

[Treijtel, 2003] describe their AmbientDB relational query processing approach for unstructured and ad-hoc networks of nodes storing data. They explicitly do not require a structured system architecture. Also, they describe a parallel query execution algorithm based on a minimal spanning tree that is constructed at the beginning of the query evaluation process from the node that received the query from an external source. However, they do assume the presence of a global schema and do not detail the construction of this spanning tree. Subsequent publications on this approach introduced a structured P2P system architecture to the model, contradicting the design goals noted earlier [Fontijn and Boncz, 2004].

[Kantere et al., 2004] present an approach of evaluating SQL-like queries in an unstructured P2P system without the assumption of a global schema being present. Since every node may use a different schema, they use rewriting techniques to adapt the query to a node's schema on every hop in the network. Since these rewriting techniques are error-prone, they introduce a configuration parameter controlling the strictness of the rewriting process. By introducing an automatic feedback on this parameter with the query evaluation success, they claim to enable the system to train itself not unlike a neural network. Furthermore, they tackle the efficiency issue inherent in unstructured P2P systems twofold: First, they use "informed" routing based on schema similarity between the nodes, and present a scheme to create new connections between nodes with similar schemata in the network to increase efficiency. Their experiments show significant efficiency improvements over flooding-based request routing, an important precondition for scalability.

## 2.5 Summary and Conclusions

In this non-original chapter, we have defined our concept of a distributed storage system and identified the competing goals of distributed storage, namely scalability, consistency and availability. DSS deployment was shown to be motivated by a higher performance / cost ratio. We have then described the system architecture and coordination method between the nodes as the main factor influencing a system's properties. Centralized, structured P2P and

unstructured P2P architectures routing were introduced as the main contenders. We have described popular data models and their respective access methods in distributed storage, namely file systems, relational databases, Key/Value stores and RDF databases. We have also reviewed the literature on the state of the art in distributed query processing for centralized as well as structured and unstructured P2P architectures.

We could see how scalability is the main recurring theme in DSS, and how economic arguments support the construction of a DSS. From our analysis of the goals for distributed systems in general listed in the both general and DSS-specific literature, we were able to select and synthesize three main goals for DSS: Scalability, which in this case refers to both being able to handle more requests for data per time unit as well as being able to store more data. Also, consistency and availability were revealed to represent the main dilemma in the design and construction of a DSS. Consistency was differentiated into several classes of consistency, and selecting one of these classes hugely depends on the systems users. Availability was described to encompass the detection and handling of errors within the system, both for the entire system as well as of individual data items. Each specific DSS carefully selects its own trade-off between availability and consistency, while striving for scalability.

Regarding the architectures, it became clear that no approach is the “silver bullet” for coordination in a DSS. Centralized nodes bear the risk of taking the entire system down with them if they should fail; structured P2P has scalability issues due to complicated node insertion and the rather inevitable formation of hot points. Finally the unstructured P2P approaches are unable to give any performance guarantees, and can also only be considered scalable if request routing does not resort to flooding.

From reviewing the different data models and access methods, structured data in any form clearly have the advantage of allowing applications to retrieve data with surgical precision, reducing application complexity as well as data traffic between the application and the DSS. However, only relational and graph databases provide complex declarative query languages able to accomplish this, and both have their issues in scalability as well as in robustness. The Key/Value stores on the other hand, designed to be able to scale with readily available

P2P technology, are currently unwilling to provide complex query support in fear of serious performance penalties.

Regarding query processing, we could see how the methods used for distributed query processing differ greatly with the underlying architecture: Centralized approaches can make use of central knowledge and thus perform effective and efficient one-time optimization. In fully distributed architectures, the presence of a global schema was seen to be problematic. Also, an one-time query optimization was shown to be no longer feasible and was replaced with continuous optimization. While the presented reference architecture is still valid in its and data structures, the flow of data and control has to be adapted for fully distributed query processing.

Furthermore, all presented methods were closely integrated with the specific underlying system architecture. However, as we have seen in the previous chapter, the network architecture defines the characteristics of the DSS. Hence, choosing different architectures, data placement schemes or routing laws might be required. However, these small changes will also inhibit the presented approaches from performing their task. Hence, each change in the environment requires changes to the query processing component. In the following chapter, we therefore define a basic and abstract environment for fully distributed query processing.

# 3 Architecture, Data and Query Models

In the previous chapters, we have discussed specific network architectures and coordination models for DSS, and shown how their properties define the performance of the system, and how its selection depends on the compromises regarding the abstract goals the system's designers are willing to make. Our survey on the state of the art on distributed query processing showed close ties to the respective network architecture. Unfortunately, this limits the broad applicability of these approaches. In this chapter, we therefore define an abstract network model for DSS based on random network graphs, which is aimed at abstract a wide range of network structures for DSS. In addition, we present the concept of a probabilistic request routing method, which is also an abstraction of the routing method that is present in a DSS to route requests to nodes where matching data is stored. The main goal for this model is the ability to describe multiple architectures and multiple levels of accuracy in the routing method. Furthermore, we discuss the issue of data placement with regards to efficient retrieval.

Also, we have seen how selective and fine-grained access models are important preconditions for retrieval efficiency, and how support for these queries is important over the whole range of DSS network architectures. Therefore, we also define a structured data and query model to be used in the remainder of this work. We use a "relaxed relational" data model, which does not require a global schema to be present. We also present a query model based on the relational algebra, and describe the both concise and expressive class of conjunctive Selection-Projection-Join (SPJ) queries on our data model.

## 3.1 Network Model

We have already assumed that the nodes in a DSS are connected using a communication medium. In theory, this enables every node to directly communicate with every other node. However, the total amount of connections increases with quadratic complexity as the number of nodes increases, which will overwhelm the network and the nodes' connectivity component quickly, as in a fully connected network the total number of connections $c$ is proportional to the number of nodes $n$ as such:

$$c = \frac{n(n-1)}{2}$$

Hence, coordination mechanisms limit the amount of nodes a single node is connected to and exchanges data with. For example, in a centralized system, storage nodes typically only communicate with the master node, and in P2P networks the total amount of connections has a fixed limit.

We therefore model the network that forms the basis of coordination in a DSS as an undirected graph $G = (N, L)$, with $N$ being a set of nodes, and $L$ being a set of bidirectional links $(n_a, n_b)$ between nodes [He et al., 2008] . From this model $G$, we can define a *neighbourhood* $H$ for each node $n$, which is a set of nodes that are connected with the current node with a network link:

$$neighbourhood(n, L) = \{n_c | (n, n_c) \in L \vee (n_c, n) \in L\}$$

Nodes are only allowed to exchange messages with nodes from their neighborhood. This model is applicable to all presented coordination mechanisms. For example, in a centralized system with a master node $c_1$ and storage nodes $s_1$, $s_2$ and $s_3$, $L$ would be $L_c = \{(c_1, s_1), (c_1, s_2), (c_1, s_3)\}$. The neighbourhood of $s_1$ is then $neighbourhood(s_1, L_c) = \{(c_1)\}$. Note how network connections have uniform weights assigned to them, which inhibits modeling different connection capacities and varying load status in this network model. However, this can be incorporated in the routing process as described in the following section.

It is important to note that no assumptions are taken regarding the network structure in this model, in particular, no small-world or scale-free properties are assumed. Rather, we only

assume a random network structure following to the Erdős–Rényi-model [Erdos and Rényi, 1960]. In this model, a network graph is constructed from a set of nodes. Each possible edge is then included with an independent probability. In order for requests to be routed to any node in the network it is necessary for the network to be fully connected, meaning all nodes can be reached from any node over a series of hops. The main reason behind this is that random networks represent the most general case for network structures, and by only requiring random network properties we do not restrict architectures and coordination models.

In DSS with a P2P architecture, such a network is bootstrapped using a distributed network bootstrap approach, which have been shown to be very robust and yet scalable [GauthierDickey and Grothoff, 2008]. For example, nodes could be passed the address of a "bootstrap node" already part of the network. The new node can now retrieve a list of neighbor nodes from the bootstrap node and request its addition to this neighbor list. This request is granted if the bootstrap node has not yet reached its neighbor upper limit as per its configuration. This process is then recursively repeated on the newly known nodes until the number of neighbors on the new node has reached the neighbor lower limit, also defined in the node configuration. If the number of bootstrap nodes is limited, this algorithm exhibits preferential attachment to create a scale-free network structure [Barabasi and Albert, 1999; Mühleisen and Dentler, 2012].

## 3.2 Coordination Model – Probabilistic Routing

For the remainder of this work, the specific approach in achieving coordination between the nodes of a DSS is not decisive. We will assume that coordination is achieved using one of the presented coordination techniques in their respective architectures. As we have seen, all techniques were based on efficiently finding an answer for the so-called *location problem*: Where should a new data item be stored, and where should the system search for data items, given a request for the corresponding identifier. As we have also seen, the different approaches enjoy greatly differing performance characteristics: While centralized systems command global knowledge and can thus find this answer in constant time with

regards to the number of nodes, structured P2P systems can give a logarithmic guarantee at least. Finally, heuristic-based systems have issues giving any guarantees on the matter. However, we are able to abstract all these methods with another heuristic:

$$P \leftarrow nextHop(n_c, k, H)$$

The result $P$ is a set of pairs that assigns all nodes from the neighbourhood $H$ of the current node $n_c$ a probability value.

$$P = (n_1, p_1), (n_2, p_2), \ldots, (p_n, p_n)$$

The values $p_*$ describe the probability that data matching the data identifier $k$ being either available on the respective neighbor node or that matching data can be found through the respective neighbor node. The sum of all probabilities has to evaluate to 1 for each node $n_c$. Formally,

$$\sum^{\forall (n_i, p_i) \in P} p_i = 1 | n_i \in H$$

Furthermore, to model the uncertainty inherent in some heuristics, a global factor $p_f$ defines the probability of the contents of $P$ being inaccurate. A value of 1.0 for $p_f$ for example is therefore equivalent to a random walk through the network.

### 3.2.1 Generic Retrieval Process

While in centralized systems the node to be visited next is always the destination node, structured P2P and heuristic-based approaches typically require multiple hops for an request to reach their destination [Stoica et al., 2001]. Keeping this in mind, we can define a solution to the location problem based on the $nextHop$ function that is agnostic to the coordination mechanism employed.

This generic retrieval process is depicted in Fig. 3.1. A request for data matching the identifier `#B` is started on node $S_2$. Assuming an error-prone routing heuristic, the picture gives the results of the $nextHop$ function for this identifier on all nodes. Probabilities of less than 6% are not pictured, but all values are given in Table 3.1.

In this table, the probabilities are given for the connection between the node in the column leading to the node in the row. Node $S_3$ has no outgoing values in this table, since the

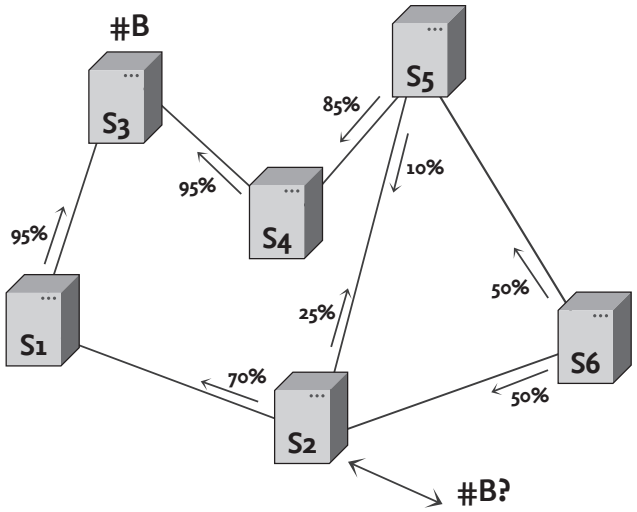


Figure 3.1: Probabilistic Routing in DSS – Example

| | $S_1$ | $S_2$ | $S_4$ | $S_5$ | $S_6$ |
|---|---|---|---|---|---|
| $S_1$ | - | 70% | - | - | - |
| $S_2$ | 5% | - | - | 10% | 50% |
| $S_3$ | 95% | - | 95% | - | - |
| $S_4$ | - | - | - | 85% | - |
| $S_5$ | - | 25% | 5% | - | 50% |
| $S_6$ | - | 5% | - | 5% | - |

Table 3.1: Routing Probabilities for Example

data item being searched for is located on this node and the retrieval process stops there. Excluding loops, the following paths are possible in this example, with their probability being the product of each probability on the path:

- $S_2$, $S_1$, $S_3$ – 67%
- $S_2$, $S_5$, $S_4$, $S_3$ – 20%
- $S_2$, $S_6$, $S_5$, $S_4$, $S_3$ – 2%

**Algorithm 1** Generic retrieval process based on $nextHop$

**Require:** Start node $n_s$, network connections $L$, data identifier $k$, hop count limit $l$

```
1:  n_c ← n_s
2:  while l > 0 do
3:      result ← searchLocal(n_c, k)
4:      if sufficient(result) then
5:          deliverResult(n_s, result)
6:          return 'success'
7:      H ← neighbourhood(n_c, L)
8:      p ← nextHop(c_n, k, H)
9:      r ← randomValue(0, 1)
10:     s ← 0
11:     for all H as h do
12:         s ← s + p(h)
13:         if s >= r then
14:             n_c ← h
15:     l ← (l − 1)
16: return 'failure'
```

Algorithm 1 describes our generic retrieval process for DSS with probabilistic routing: Requests for data include the data identifier $k$ and are started at any node that is part of the DSS. First, the node checks whether it is able to serve the request itself with the locally stored data. If sufficient data is found, the results are delivered to the node where the request had originated from, as the client application is presumably waiting for results there (Line 3ff). Results can then be delivered by tracing back the path the operation has taken through the network, with the possibility of eliminating loops beforehand.

If that is not the case, the node uses the $nextHop$ function to determine the probability for each neighbour that it is either able to handle the request or is at least better suited to forward the request to its destination. From these values, a weighted random decision is taken (Line 8ff). Since infinite loops are possible in probabilistic routing, a limitation of the number of

hops the process is allowed to make is also included in the generic algorithm. If the process should hit this step count limit, the process was unable to find matching data and finishes with a failure. In addition, there is a possibility that a retrieval process is started with a data identifier for which there is no data stored within the DSS at all. However, for the sake of simplicity in the network and coordination model, we have not provided for this case. The non-availability of data for a particular key is therefore defined as the retrieval process being terminated by the step count limit.

### 3.2.2 Retrieval Guarantees and Accuracy

To show the versatility of our abstraction, we now present examples on how a specific network architecture and coordination mechanism may implement the abstract model.

For DSS with a centralized or structured P2P coordination method, the $nextHop$ function enjoys certainty regarding which node to visit next. In this case, the probability distribution $p$ evaluates to 1.0 for the correct next hop and 0.0 for the other neighbor nodes. In this case, the weighted random decision taken by the generic retrieval process degenerates to a fixed decision towards this node. For example, if the DHT decides the next hop to be $S_1$ from a neighbourhood of $H = \{S_1, S_2, S_3\}$, the result of $nextHop$ would be $p = \{(S_1, 1), (S_2, 0), (S_3, 0)\}$. In this case, regardless of the employed randomness, the algorithm selects $S_1$ as next node to be visited deterministically. In these architectures, we are then also able to provide the guarantee that data present in the DSS will also be found by our algorithm, if the path length through the network is smaller than the step limit.

In a centralized architecture, all operations are coordinated by a master node as described in Section 2.2.1. This master node also holds sufficient information to determine where in the network a data item is being stored. To reflect these properties, our model can be configured as follows: The network consists of a central node $C_1$ and a number storage nodes $S_*$, $N = \{C_1, S_1, \ldots, S_n\}$. Connections between the nodes exists such that the $C_1$ is connected to all storage nodes, and no further connections exist, $L = \{(C_1, S_1), \ldots, (C_1, S_n)\}$. This way, the neighbourhood of $C_1$ would be the set of storage nodes $\{S_1, \ldots, S_n\}$ and the neighbourhood of each storage node would only be $\{C_1\}$. Furthermore, the central node holds a local data structure $G$, where a mapping $(k, n)$ between a routing key $k$ and a storage

node $n$ determines where data with this key are stored. The probability distribution $p$ of the routing function on all nodes can now be given as follows:

$$p(n \in H) = \begin{cases} 1 & n = C_1 \\ 1 & (k, n) \in G \\ 0 & otherwise \end{cases}$$

Contrary, a structured P2P network does not possess a central node. Here, the set of nodes is therefore $N = \{S_1, \ldots, S_n\}$. However, the global law such as a ring-shaped DHT dictates the network structure as described in Section 2.2.2. Here, the set of connections $L$ is constructed such that it correlates with the hash value $k$ at the beginning of the value range the node is responsible for. During network construction, all neighbour nodes in $H$ are queried for that value, and each node keeps a local data structure $G$ with these values such that $G = \{(n, k_n) \forall n \in H\}$. Using these values, the routing function can now determine the node closest to the hash value for which data is currently being searched. Data retrieval for a key $k$ is now a matter of finding the node closest to that value from $G$. The probability distribution $p$ for our model is therefore:

$$p(n \in H) = \begin{cases} 1 & (n, k) \in G \wedge min(k - k_n) \forall (n, k_n) \in G \\ 0 & otherwise \end{cases}$$

The more interesting case is where a heuristic is used to improve a random walk through an unstructured network as described in Section 2.2.3. Here, the heuristics influences the routing decision towards the neighbor node it sees best suited to continue with the request. While it is impossible to describe all possible heuristics here, we will assume for the following that this has a "beneficial" tendency. A heuristic is regarded to be beneficial, if on average the probability distribution created by $nextHop$ leads to the best candidate from the neighbor list being selected. Formally, a routing heuristic is beneficial if its error probability $p_f$ is in the range $[0, 0.5[$. Through repetition, a beneficial heuristic will lead to a high probability of the request reaching its destination. It should be noted that in unstructured architectures, this method cannot lead to deterministic behaviour and the success of retrieval operations cannot be guaranteed any more. However, though its configuration with the $p_f$ parameter, it can be configured to appropriately model the performance of the specific heuristic used.

Therefore, our model is appropriate to abstract from at least the three presented network architectures, and possibly many more. In the remainder of this work, we use only this network model. If an algorithm is able to perform its task on this abstract model, it is also able to do so on the abstracted architectures. Thereby, we avoid the common pitfall we noted in Section 2.4, which consisted of tying an algorithm closely to a specific network architecture. While this approach can give some performance improvements and allows exploitation of the inner workings of the coordination model, it also makes the transfer of an approach to another architecture hard, which is precisely what we are trying to avoid here.

### 3.2.3 Stochastic Analysis

To determine the theoretical performance of routing heuristics, we will now perform a stochastic analysis of the retrieval process. To this matter, we will describe the average case performance of the algorithm. Consistent with distributed systems research, the unit of cost for this analysis will be hops, that is the amount of transitions of the operation between nodes [Peleg and Pincas, 2001; Tang et al., 2008].

Since a request can be started at every node and results can be on any node in the network, the cost for any retrieval operation is at least the length of the shortest path between the nodes in the network. In the average case, this distance is the average path length in the network. Disregarding the possibility of the network graph having small-world or scale-free properties, we assume the average path length in random networks as our average distance from start to target node. The average path length in a random network $l_{ER}$(and also the average distance between nodes) is calculated as follows [Fronczak et al., 2004]:

$$l_{ER}(N, \langle k \rangle) = \frac{\ln N - \gamma}{\ln \langle k \rangle} + \frac{1}{2}$$

with $N$ being the number of nodes, $\gamma$ being the Euler-Mascheroni constant ($\approx 0.5772$) and $\langle k \rangle$ being the average connectivity in the network (equivalent to the average number of neighbor nodes).

For this analysis, a heuristics-based DSS with a beneficial routing heuristic as defined in the previous section is assumed with $p_f$ in the range $[0, 0.5[$. For every step on the way from the origin to the destination node, three outcomes of the heuristic-supported routing process

are possible: Positive, where the operation got one step closer to its destination, Neutral, where the amount of steps remaining is unchanged, and Negative, where the operation now is one step further away from the destination. Since network connections are defined to be bidirectional, a step in the wrong direction can add at most one additional step to the remaining path length.

From observing the routing heuristic, we assume to have determined its $p_f$ value, but now we have to discern between the positive and the neutral/negative case. However, this distribution between neutral and negative outcome is unknown, we therefore introduce a second parameter, $p_n$. The probabilities for each case are thus as follows:

- $p(positive) = 1 - p_f$
- $p(neutral) = p_f * (1 - p_n)$
- $p(negative) = p_f * p_n$

For a single step in the network, the total impact $i$ on the remaining path length is calculated as $i = -(1 - p_f) + (p_f * p_n)$. If the assumption of $p_f$ being at most 0.5 holds, we can see that $p_n$ has to be 1 in order for the improvement $i$ to evaluate to 0. However, it is unlikely that every mistake adds another step to the operation's path, and hence we can safely assume $p_n$ being smaller than 1. If this assumption holds, the improvement $i$ is always negative. Consequently, every routing operation will – on average – bring the retrieval process closer to its destination.

The expected value for the average hop count to retrieve an arbitrary element from the network is then the fraction of the average path length by the absolute value of the expected reduction of the remaining path length per hop.

$$hops(N, \langle k \rangle, p_f, p_n) = \lceil \frac{l_{ER}(N, \langle k \rangle)}{|-(1 - p_f) + (p_f * p_n)|} \rceil$$

Table 3.2 lists the result of this calculation a network size of 10,000 nodes and an average connectivity of 10. The average path length for a random network with this characteristics is 6 hops according to our formula. In the table, we have calculated the expected number of hops with probabilistic routing for various $p_f$ and $p_n$ settings. We can see how only the

$p_f$ value has a considerable influence on the hop count and overhead between average path length and hop count created through the use of probabilistic routing. The decision whether the incurred overhead is acceptable can now be weighted. Through the analysis of a specific coordination method, a DSS implementation can now choose whether the trade-off between coordination effort and routing errors is acceptable. We present two experiments aimed at validating our analysis in Section 5.3.

| $p_f$ | $p_n$ | $i$ | hops | overhead |
|---|---|---|---|---|
| 0.1 | 0.25 | -0.88 | 7 | 1 |
| 0.1 | 0.5 | -0.85 | 8 | 2 |
| 0.1 | 0.75 | -0.83 | 8 | 2 |
| 0.2 | 0.25 | -0.75 | 8 | 2 |
| 0.2 | 0.5 | -0.7 | 9 | 3 |
| 0.2 | 0.75 | -0.65 | 10 | 4 |
| 0.3 | 0.25 | -0.63 | 10 | 4 |
| 0.3 | 0.5 | -0.55 | 11 | 5 |
| 0.3 | 0.75 | -0.48 | 13 | 7 |
| 0.4 | 0.25 | -0.5 | 12 | 6 |
| 0.4 | 0.5 | -0.4 | 15 | 9 |
| 0.4 | 0.75 | -0.3 | 20 | 14 |
| 0.5 | 0.25 | -0.38 | 16 | 10 |
| 0.5 | 0.5 | -0.25 | 24 | 18 |
| 0.5 | 0.75 | -0.13 | 48 | 42 |

Table 3.2: Probabilistic routing – Average hop count expectation

To come back to our stochastic analysis of the average amount of hops required to route a request from the node it was created on to the node storing matching data, we have described the average amount of hops required to perform this task in the presence of an potentially unreliable routing heuristic. When we review the formula to calculate the average amount of hops required to retrieve a single data item, we can see that a logarithmic complexity with

regard to the number of nodes is present. This is consistent with previous research on fully distributed coordination models [Loguinov et al., 2003; Giakkoupis and Hadzilacos, 2007]. We therefore regard our abstraction of the coordination subsystem to be both simplistic yet potentially efficient and can now continue to introduce our data and query model.

## 3.3 Data Model

As we have seen in Section 2.3.2 and Section 2.3.3, the relational data model provides the highest amount of consistency for the stored data. Through its global schema, data is highly structured and can be efficiently distributed through horizontal and vertical distribution with centralized coordination. Also, queries over data in this model can be validated through comparison of the request with the schema. However, we have also seen that one of the motivations for research on Key/Value DSS was the complexity inherent in distributing the relational model. If the coordination model is no longer centralized, the DSS would also need to distribute the schema information. The methods available for this distribution (replication, partitioning etc.) cannot guarantee the schema being up-to-date on every node, which would be a requirement for data consistency. Hence, agreeing with the NoSQL approach, we will not require a schema to be present in our data model.

Rather, we remove the concept of a relation, a set of tuples with matching headers. The bulk of the data is now only a set of tuples, with potentially each tuple having its own header. This data model is directly equivalent to the Key/Value model, with its main advantage of being able to be distributed efficiently. Formally, a tuple is a function $t : K \rightarrow \{V \cup \perp\}$, that maps a subset of keys $k \in K$ to a value $v \in V$. In the case where $t$ does not contain a mapping for a key, $\perp$ is returned.

For the sake of modeling simplicity, we express all tuple values $V$ as string literals. Should different data types be desired, these could be added as a third entry to each Key/Value mapping. Both the relational as well as the RDF data can be expressed within this model, as we will show below. Furthermore, this model is equivalent to the data model used in the Pig data analysis system. Here, complex queries in a procedural language are translated into a set of (distributed) Map/Reduce operations [Olston et al., 2008].

Manufacturer

| Name | City |
|---|---|
| BMW | Munich |
| Daimler | Stuttgart |

City

| Name | Inhabitants |
|---|---|
| Stuttgart | 607000 |
| Munich | 1330000 |

Table 3.3: Relational Data – Example

To express relational data in the Key/Value model, we add the name of the relation to each tuple as such: For every relation in the database $R = (B, H_R, i)$ with $B$ being the set of tuples, $H_R$ being its common header, and $i$ being the identifier of the relation, we extend each tuple $t$ from $B$ with a new mapping to the relation identifier such that $t('isA') = i$. The union of all tuple sets $B$ from all relations is then the result of the transformation. Should same-value attributes names occur in multiple relations, the relation name is prefixed to the attribute name to remove ambiguity.

For example, if we transform the relational data from Table 3.3 into our data model, the result would be the following set of tuples:

```
{
{('isA','Manufacturer'),('Name','BMW'),('City','Munich')},
{('isA','Manufacturer'),('Name','Daimler'),('City','Stuttgart')},
{('isA','City'),('Name','Stuttgart'),('Inhabitants','607000')},
{('isA','City'),('Name','Munich'),('City','1330000')}
}
```

A very similar method can be used to express RDF graphs in the Key/Value data model: For every triple $(s, p, o)$ in the graph's serialization, we create a tuple $t$ such as $t('s') = s$, $t('p') = p$ and $t('o') = o$.

### 3.3.1 Local Storage Interface on Nodes

To write data in this model to nodes and also read it again, we also define a deliberately primitive storage interface to be present on each node. At the same time, we do not make further assumptions about the algorithms, data structures, and storage devices used to physically store this data. As we have seen, our data model is based on sets of schema-less tuples. Therefore, the largest coherent units of data in this model are tuples themselves. The signatures of the two methods of the local storage interface are:

- $put(t)$
- $T \leftarrow read(k, v)$

To store tuples on a node, the $put(t)$ is used. $t$ refers to an arbitrary tuple with its set of key/value bindings. The $put$ operation does not give a return value. To read data from a node, the $read(k, v)$ operation is used. $k$ refers to a key of any tuple previously stored, and $v$ refers to a value. $read$ returns a set of tuples $T$, which may be empty if no matching tuples are found. Formally, if we assume $D$ to be the set of all tuples stored on a particular node, $read(k, v) = \{d \in D | d(k) = v\}$. Furthermore, $v$ can be replaced with a wild card marker $*$, in which case all tuples that contain the key $k$ are returned.

## 3.4 Data Distribution Scheme

An important aspect for our model is how tuples are placed inside the storage network. Since no schema connects tuples any more, placement can be decided for each tuple separately. Since the only mode of access to tuples is by a single-valued routing key, we adopt the distribution scheme proposed in [Cai et al., 2004]. There, each tuple is stored several times, each time with a different tuple entry key as routing key. This enables retrieval operations for

tuples by each entry key, a precondition for selective access. Of course, this creates a storage overhead linear to the number of entries in the tuple. However, since this overhead is limited by the tuple length, which can be safely assumed to be far lower than the total amount of tuples, the multiple copies created by this method only contribute a linear complexity to the effort required to store all tuples.

While we are in theory now able to retrieve the stored tuple by any of their entry keys, the main question remains. On which node should the tuples be stored? In a storage network using a centralized coordination mechanism, the central node can answer this question. This makes a balanced distribution of tuples according to storage and query load possible. In contrast, in a coordination mechanism lacking a central node such as distributed hash tables or heuristic-based approaches, the decision where to store is also distributed. For example, in a DHT the global law dictates a mathematical relationship between routing key and network node that is responsible for storing the data item. In DSS using a heuristic-based coordination mechanism, a number of methods to determine the placement of tuples according to its routing key are conceivable.

To avoid restricting our model to a specific coordination mechanism, we will assume the presence of a *data distribution scheme*, but make no further assumption about it. In the most general case, single tuples are randomly placed inside the storage network, a notion that also supports our stochastic analysis of the single-item retrieval mechanism described in Section 3.2.3. Also, since the data placement problem in a DSS has been shown to be $\mathcal{NP}$-complete [Gribble et al., 2001], achieving a perfect data distribution with regards to efficient retrieval is unrealistic from the outset. Several heuristics have been proposed to create a near-optimal data placement in a DSS [McClean et al., 1991]. However, these methods rely on central control for optimization heuristics such as random improvement or simulated annealing, and are thus not applicable in our environment. Therefore, we assume a random data placement scheme as the most general case for the remainder of this work.

A closely related and more serious issue arises when multiple data items are to be stored with the same routing key. This is very likely, for example, in our data model given in Section 3.3, the tuple key `isA` is occurring in each tuple. According to the data placement scheme outlined above, each tuple would be stored once with this value as routing key.

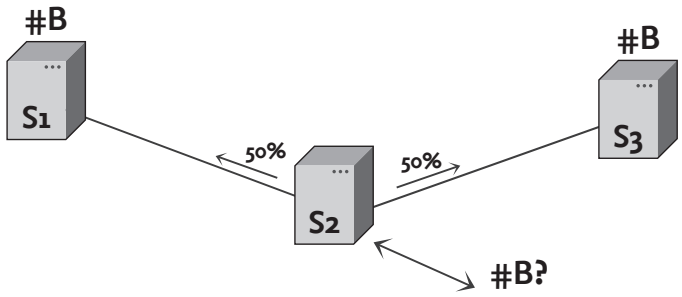


Figure 3.2: Data Placement with Equivalent Routing Keys – Example

At the same time, retrieval operations for this key would expect all matching tuples to be returned. With a random placement strategy, this retrieval operation would degrade to a scan of the entire DSS, which is not efficient. This situation is depicted in Fig. 3.2. Here, a retrieval request for the routing key `#B` is started on node $S_2$, which matching data both being available at $S_1$ and $S_3$. The perfect routing heuristic $nextHop$ on $S_2$ now yields $p(S_1) = p(S_3) = 0.5$. To retrieve matching tuples from both nodes, the retrieval operation would have to make 3 hops, which is equal to the number of nodes in the network and not desirable at all.

An alternative solution is to extend the placement scheme to store tuples with equivalent routing keys on the same node. This placement scheme is for example the default mode of operation in a DSS using a DHT as coordination mechanism. However, this represents a threat to the scalability of a system implementing our model as well. If all or a significant fraction of the entire set of tuples to be stored in the DSS contain a single tuple entry key, the local storage of the node storing tuples for this key would soon be overloaded [Battré et al., 2006].

It is therefore necessary to provide a method which both enables efficient retrieval as well as allowing overflowing nodes to gracefully offload excess data to other nodes. Again, in the case of a centralized system this can be achieved using central load monitoring and the partitioning of data stored using high-frequency keys between several storage nodes. In ring-based DHTs, this data is often moved to the successor nodes [Wang et al., 2006]. In heuristics-based system as well as in our coordination abstraction, a different method has

to be used. To achieve efficient retrieval, one possibility is to create *key locality* by storing tuples with equivalent routing keys in a very limited subset of the network. In our abstraction, a possible algorithm to create this locality would be first to locate the node already storing data with the routing key of the tuples to be stored, typically by creating a read operation. If such a node is found, the first choice would also be to store the new data there. As this node comes close to exhausting its storage capacity, new data with this key is then moved to the neighbor nodes, with the original node keeping track of which neighbor nodes the excess data was moved to. Incoming retrieval operations can then either be forwarded by modifying the result of the $nextHop$ function, or by being given a list of neighbor nodes where additional matching data is located.

As stated before, the retrieval operation is allowed to exhibit a certain probability of error. If too many errors occur, the node storing data for a single key might not be found, even though tuples with matching routing key are present in the network. In this case, the new data item would be placed on an arbitrary node. This creates the aforementioned problem of the new data being either not accessible at all or requiring an extraordinary amount of hops to locate. To mend this issue, we have presented an algorithm inspired by the brood sorting used by some species of ants in our previous work [Mühleisen et al., 2011c]. In a nutshell, each node periodically sends out probes with a histogram (routing key / number of stored tuples) of the locally stored data. Other nodes receive this data, and compare this histogram to the locally stored data and either move the data from the node the probe originated from into their local storage or move the locally stored data to the probing node.

For the remainder of this work, we can therefore safely assume a random data placement scheme for each routing key, with tuples being stored by the same routing key on the same node. We have presented a conceptual method to mend the problem of overflowing nodes, but will not consider the issue further for the sake of simplicity.

## 3.5 Query Model

As we have now described how structured data is expressed in our generic data model, we can continue with the description of the fine-grained access to data stored in this model through complex queries. Before we can go into the details of complex query processing, we must first define what we regard to be a complex query. While the literature does not contain a usable definition, there seems to be consensus on what kind of operations can be part of a complex query.

Even though we have removed a crucial part from the relational data model – the concept of a relation – we are using the complex query operators supported by relational databases as the basis for our query model, as they provide a time-proven starting point. The relational algebra first proposed by [Codd, 1970] used set theory to describe their operations, namely *selection*, *projection*, *cross product*, *set union* and *set difference*. Together with the *rename* operation shown to be necessary in [Hall et al., 1975], these operations form a fundamental nucleus of operators and also provide the entire expressiveness of the relational algebra. Furthermore, the relational query algebra can also provide a basis to support additional data models and query languages. For example, queries in the RDF graph query language SPARQL can be and typically are translated into SQL queries [Elliott et al., 2009]

Even though the common query language SQL does not fully implement the relational model, its `SELECT` statement does include further operators to be considered [Melton and Simon, 1993]: In addition to the operations already presented, SQL defines the following operations: Multiple relations may be joined together by a number of different join methods. In a join, values from multiple relations are recombined into a new relation, often combining the result tuples by a join criterion. We will consider the *equijoin* as a popular example for join operations further. Furthermore, SQL supports the modification of the result set with the *group by* and *order by* operators. However, these operators can always be applied on the final result set and are thus not considered further. Furthermore, these operators have been shown to only possess subtle differences to join processing [Bratbergsengen, 1984]. Consistent with previous literature on complex query processing, we therefore focus on the class of *conjunctive SPJ-Queries*, which represent a more tractable set of operators, which

nonetheless also exhibit the complexity inherent in full-blown SQL query processing [Tudor, 2007; Chekuri and Rajaraman, 2000]. SPJ queries contain the selection operator, which reduces a set of tuples based on a selection criterion, a projection operator, which removes mappings from individual tuples, and a join operator, which combines tuples by common attribute values.

We will adapt the operators selection, projection and join for our query model and discuss each of these operators considering our non-relational data model as defined in the previous section. To be able to perform query processing, we need to define the correct result for each operator, and their general input/output structure.

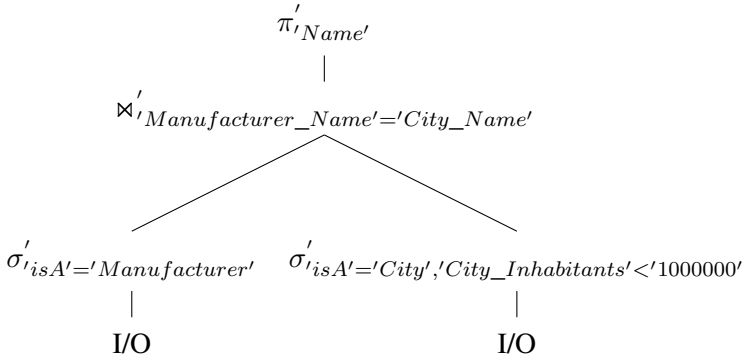


Figure 3.3: Example Query – Tree Representation.

To give an example of a complex query expressed in the model we have defined in this section, we will formulate a query retrieving car manufacturers from cities with less than one million inhabitants from the data converted from the relational format in Section 3.3. The query to retrieve this information is given in Fig. 3.3 in a tree representation. Data flows through this tree from bottom to top, as soon as the I/O processes finish, they deliver their results to their parent nodes. In this example, the left selection will deliver the two tuples describing the manufacturers to the join, and the right selection one tuple describing the city with less than 1M inhabitants. As soon as both selections are finished, the join can evaluate, and finally the projection to the company name, which will yield a tuple set with a single tuple with one attribute mapping, in this example $\{\{('Name','Daimler')\}\}$. We will omit the quotation marks on attribute names and values for the remainder of this work.

As a data structure, queries are represented as hierarchical trees of operators. Each *operator* holds a reference to a single parent operator, and a list of references to child operators. Furthermore, operators contain a Boolean flag that marks their evaluation state as well as a set of tuples as the result of their application. Formally, an operator $o$ is defined as a 5-tuple $o = (p, c, f, r, e)$, with $p$ being the parent operator, $c = (c_1, c_2, \ldots, c_n)$ being an ordered list of child operators, $f$ being the flag that marks an operator to be fully evaluated, and $r$ as the set of tuples as the result of this operator. In order to refer to an entire query $q$, it is sufficient to give a reference to its root node $o_r$, since all operators that are part of the query tree can be reached by recursively traversing the $c$ list.

$e$ is the operators *evaluation function*. The result of the evaluation function is available as soon as the child operators finish. The specific parent operation (selection, join, etc.) can then calculate the result of its evaluation function $e$, as soon as all child nodes have been marked as evaluated and their results being present. As soon as the root node has been evaluated, the query is fully evaluated. On the other end of the tree, leaf operations in the query tree have the capability of accessing the stored data, e.g. by reading from persistent storage.

In the following, we will now express the operations identified to be representative for complex query evaluation within our operator model. Here, it will be sufficient to describe the evaluation function $e$ for all operators.

### 3.5.1 Selection

Relational selection is defined on a relation $R$ as $\sigma_{a\theta p}(R)$, with $a$ being an attribute name, $\theta$ being an comparison operator, and $p$ being either another attribute name or a constant value. The result of the selection are the tuples with a mapping for $a$, for which the comparison with $p$ evaluates to `true`. For example, to select all cities with more than a million inhabitants from the relation "City" given in Table 3.3, one could use the following selection: $\sigma_{'Inhabitants'>1M}('City')$. In our data model, we replace the relation $R$ from the above definition by an unstructured set of tuples $T$. However, in this case we can no longer be certain that all tuples in $T$ contain a mapping for the selection attribute $a$. Hence, we redefine the selection as $\sigma'_{a_1\theta_1 p_1, a_2\theta_2 p_2, \ldots a_n\theta_n p_n}(T)$, allowing multiple selection criteria to be

set for a single selection operation. Furthermore, we also redefine the selection result, such that tuples that lack a mapping for either $a$ or $p$ (if $p$ is an attribute name) are not included in the result set. The $e$ function for the selection operation thus takes the result from the first and only child operator, applies the selection criteria, and then passes the result on to its parent. Formally, the result set $r$ of our selection is defined as

$$t \in c_1.r \wedge (\theta_1(t(a_1), p_1) \wedge \ldots \wedge \theta_n(t(a_n), p_n)) \Leftrightarrow t \in r$$

### 3.5.2 Projection

The relational projection removes attributes from all the tuples in a relation, formally $\pi_{a_1,a_2,\ldots,a_n}(R)$. The tuples in the resulting relation then only contain the projection attributes $P = \{a_1, a_2, \ldots, a_n\}$. Similar to the selection, we replace $R$ by the unstructured set of tuples $T$ and redefine the projection as $\pi'_{a_1,a_2,\ldots,a_n}(T)$. As with the selection, tuples in the input set that do not contain a mapping for one of the projection attributes will be omitted from the result set. The $e$ function for the projection operation thus takes the result from the first and only child operation and applies the projection attributes to create its result set. Formally, the result set $r$ of this projection is defined as

$$t \in c_1.r \wedge \forall a \in P | t(a)! = \perp \Leftrightarrow$$
$$t_r \in r \wedge \forall a \in P | t_r(a) = t(a) \wedge \forall m \notin P | t_r(m) = \perp$$

### 3.5.3 Equi-Join

As mentioned, we use the relational equi-join (an $\theta$-join with only equality operators) as an example as to how more complex relational operations may be evaluated within our reduced data model. The relational equi-join compares the values of the given attributes in the input relations. We define our equi-join operation as follows: $T_1 \bowtie'_{a_1=a_2} T_2$, with $T_1$ and $T_2$ being sets of tuples and $a_1$ and $a_2$ being attribute names. $a_1$ denotes the join attribute from the first input set $T_1$, $a_2$ is an attribute from $T_2$. The main issue here is again how tuples in the input sets that have no mapping for the join attribute should be handled. Consistent to the selection operator, these tuples are ignored. The $e$ function for the equi-join thus takes the result sets

from the first and second child operation and returns joined tuples with equal values for the join attributes. Formally:

$$\forall t_1 \in c_1.r \forall t_2 \in c_2.r(t_1(a_1) = t_2(a_2) \wedge t_1(a_1)! = \perp) \Leftrightarrow$$

$$t_r \in r \wedge \forall k_1 \in keys(t_1)(t_r(k_1) = t_1(k_1)) \wedge \forall k_2 \in keys(t_2) \backslash a_2(t_r(k_2) = t_2(k_2))$$

## 3.6 Summary

In this chapter, we have laid the foundations for architecture-agnostic distributed query processing: We have presented a model for a network structure based on the very general concept of random networks as well as a fully distributed network construction algorithm. Based on this model, we have described our probabilistic request routing approach, which does no longer require the routing function to deliver exact results. This has the main advantage of removing the need for complete knowledge of the location of data inside the network or inflexible global laws. Furthermore, we have presented an abstract algorithm to retrieve any data item from any node that is part of the storage network. The issue of data placement has also been discussed, and we concluded on the need for key locality, if traversing the entire storage network in search for data is not an option. Through an average-case stochastic analysis, we were able to predict the relationship between the error rate of the routing function and the additional costs. Furthermore, we have defined a structured data model, which does not require a global schema. We have continued to discuss the distribution of data expressed in this model in a DSS, and concluded on a need for locality in data placement for efficient and complete retrieval results. We have also presented a query closely following the relational query model that supports selection of subsets of data, projection of tuples to reduce their size, and equi-joins of two sets of tuples based on common attributes. In the following chapter, we will use the models defined in this chapter to describe architecture-agnostic distributed query processing.

# 4 Distributed Query Processing with Mutable Moving Query Plans

In the previous chapters, we have seen how support for complex queries in a DSS is highly desirable, since they allow us to answer questions on the entire stored dataset without having to retrieve several potentially large intermediate results. In order to gain independence of specific network architectures and coordination methods, we have described an abstract network architecture that represents a lowest common denominator of several popular network architectures. We have also presented a query and data model, which do not rely on any central knowledge, as this central knowledge represents a potential threat to the system's distributed nature. Furthermore, our review of the state of the art in fully distributed query processing has shown that distributed query optimization is best performed through continuous improvements. In this chapter, we now investigate the question whether it is possible to efficiently evaluate these complex queries within our abstract network model. The presented method is not intended to be directly implemented, since it is only based on an abstraction. However, it is valuable in providing a baseline algorithm to determine lower complexity bounds and can also serve as the basis for an implementation for a specific network architecture. Furthermore, we can test our assumption whether probabilistic routing and key locality in data distribution are sufficient for logarithmic complexity in distributed query processing.

We structure this investigation according to the generic DQP architecture and process presented in Section 2.4. However, to reflect the lack of a schema and the continuous optimization process, we have adapted the generic architecture shown in Fig. 4.1. Here, the query received by the system is first *parsed* into a internal tree-like structure and then

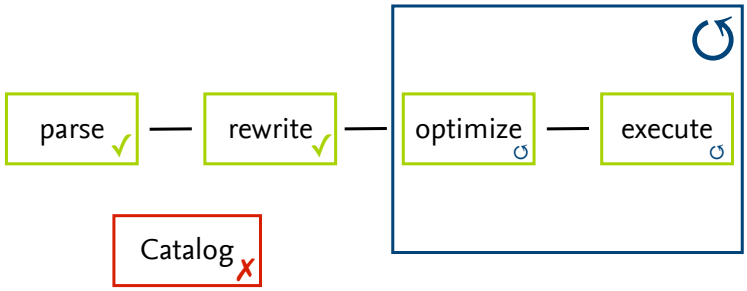


Figure 4.1: Distributed Query Evaluation with Continuous Optimization

*rewritten* to an internal operator structure. The process of *optimizing* and *executing* this operator structure is no longer a single step, but constantly repeated to adapt the process to new information as it becomes available. Furthermore, the *Catalog* component has been removed for the mentioned reasons. As we can see, parsing and rewriting a query into the internal query model is unchanged, allowing us to re-use previous work and not consider these steps further.

The main focus here is therefore optimization and execution. To achieve these tasks, we have devised the concept of Mutable Moving Query Plans (MMQP), which perform query processing as part of a journey through the network, permanently seeking to optimize the query at hand through re-formulation as well as operator reordering. At the same time, operators in the query tree are evaluated, as soon all data required by them has been located. This concept is loosely based on two previous ideas and combines them: [Papadimos and Maier, 2002] have described the idea of *mutant* query plans for distributed XML databases, while [Avnur and Hellerstein, 2000] have proposed *movable* query plans, that are passed from node to node, accumulating partial results until the query is fully evaluated.

In this chapter, we list the assumptions we have taken in the design of MMQP, present a procedural overview of our method, and describe the cost model used to assign costs to alternative query execution plans. We discuss evaluation efficiency and result set completeness along with methods to recover from failures. We end this chapter with a discussion of the efficiency of the approach presented here and give an average-case stochastic analysis aimed at predicting the evaluation costs that are to be expected.

## 4.1 Assumptions and Preconditions

Before we go into the details of our MMQP approach, we first list the assumptions we have taken in its design. As mentioned, we try to achieve distributed query processing. We assume the network model described in Section 3.1. As part of this model we also assume the presence of a usable routing heuristic. In particular, the routing heuristic is required to have a failure probability of less than 50%, as we are otherwise unable to reach stored data in a reasonable number of steps, as discussed in Section 3.2.3. Apart from these, no further assumptions are taken for the network architecture, in particular allowing the addition and departure of nodes at any point, even during query processing. We also assume data according to the data model presented in Section 3.3 being present in the network and – more importantly – that the data placement scheme inside the network exhibits key locality as outlined in Section 3.4. Also, replication is assumed to be present to protect against node failure and load issues, but that the replicated data is transparent to the network architecture and local storage operations. Finally, data may be added, removed or moved at any time, including during the process of evaluating a single query. As mentioned, we also assume queries being present in the tree model as described in Section 3.5, only containing the three operators Selection, Projection and Join.

More importantly, we assume an ability to execute program code on the nodes in the DSS. More specifically, we assume nodes to be able to react on messages transmitted to them from one of their neighbor nodes by executing a custom program as defined below in Section 4.4. This execution context allows us to send the query through the network without tracking or controlling this process from a single location. Otherwise, we would have no choice but to assemble all data relevant to the query prior to execution, which is also known as "data shipping”. However, one of the goals of complex queries in our model is to significantly reduce the amount of transmitted data through the use of selections or join operations with other intermediate data. Therefore, the described local execution context is required. This assumption also implies that the nodes in the DSS are under the control of a single organization, since non-related organizations would be highly unlikely to allow the execution of code on their machines. Therefore, our approach is for example not suitable to

public P2P infrastructures. On the other hand, this ownership model also allows us to not consider security implications or malicious nodes, and makes our assumption of a usable data distribution scheme being present more realistic.

Two important aspects to complex query processing from a traditional database perspective are the correctness and completeness of query results [Ramamritham and Chrysanthis, 1996]. These aspects can be compared to the consistency introduced as one of the competing goals in Section 2.2. In our environment, correctness of query results can be guaranteed, since a deterministic process using defined operators is used to produce them. This distinguishes our approach from Information Retrieval in general.

Continuing our argumentation on consistency, and how compromises on consistency are required to achieve the other goals, we believe it is also necessary to make compromises regarding query result completeness. Fortunately, our network model together with the data distribution scheme enables us to adjust our model to completeness requirements. If completeness of query results is required, adopting for example a data distribution scheme (see Section 3.4) of same-key data being stored on a single node along with a error-free routing method for example being present in centralized networks, we are able to guarantee that – given a hop count limit sufficient to traverse the entire network – the query results will be complete. However, we will not require these two preconditions, since they would severely limit the range of network architectures our approach would be applicable to. In situations where the two preconditions do not hold, we are unable to provide any completeness guarantees, and only stochastic guarantees can be given for even partial query evaluations. We discuss the stochastic properties of our approach in Section 4.6.

## 4.2 Procedural Overview

Our concept of query evaluation is performed as follows: Within the DSS, any node receives a complex query for the data stored in the entire network by an application or user. The node will then parse the query into an internal tree representation as described in Section 3.5. Without initial optimization, the query is handed over to the execution component, which first searches the local data for tuples matching the basic I/O operators in the query plan. If

tuples are found, they are fed into the operators, which can then trigger their parent operators to also evaluate as described.

As described in Section 3.3.1, local I/O operations in our model are modeled as retrieval operators by a single key and (optional) value from local persistent storage, yielding a set of tuples. To continue our example from Section 3.5, the query tree with the local retrieval operator $read(k, v)$ is given in Fig. 4.2.

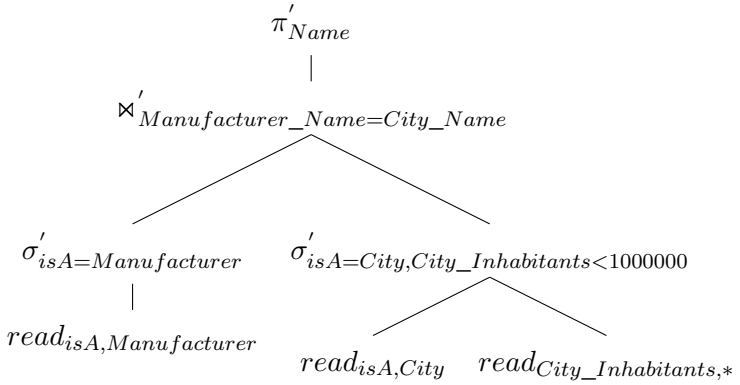


Figure 4.2: Example Query – Tree Representation.

Then, the set of result-equivalent execution alternatives is computed, each with a specific order of basic read operator execution (Similar to the operator "ToDo" list described by [Rösch et al., 2005]), a more formal definition follows in Section 4.4. These alternatives are then ranked according to a cost model based on the routing heuristic, and a selectivity and distance estimation as will introduce in Section 4.3. Based on this ranking, the best query plan based according to the local knowledge is selected for further processing. The next operator to be evaluated is then selected from the operator list, and the entire query along with the partial results inside the operators is forwarded to the node which has the highest probability of delivering results similar to our single-objective retrieval algorithm (Algorithm 1).

A data flow diagram of this process is pictured in Fig. 4.3: A complex query is started on the leftmost node, which creates the abstract query tree representation as well as the operator ordering (denoted by the numbers in circles). At this point, the leftmost read operation is to

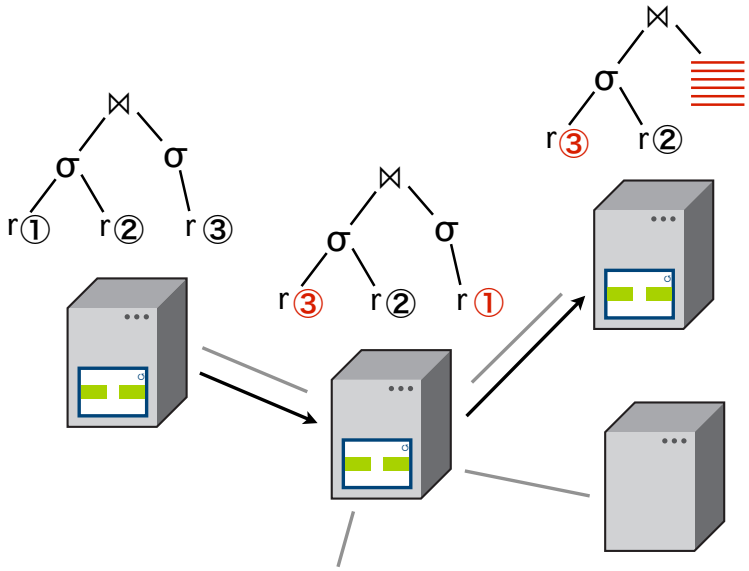


Figure 4.3: Mutable Moving Query Plans – Conceptual View

be executed first. Since this node does not contain any data matching the read operations, the query execution is forwarded to next node. According to data retrieved from the routing heuristic, this node then decides to switch the operator execution order between the leftmost read operation and the rightmost read operation. Since this node also does not store relevant data, the query is forwarded again to the top right node. Here, data matching the read operation with the highest priority is found, replacing the read operator. Since its parent operator – a selection – has no other non-evaluated child nodes, it can also be evaluated. However, its parent join operation cannot be evaluated yet, as it has to wait for the results of the other selection. In this form, the query is now forwarded further, until the root node has been evaluated as well.

The reasons for the changed operator order in our example is given in Fig. 4.4. Here, query processing is again started on the leftmost node. In this example, only two read operations are included in the query tree, one for data with the key # and one for data with the key $\star$. At this point, the node calculates the routing probabilities for both keys, and finds the probability to find data for # on its only neighbor node is far higher than finding data for $\star$.

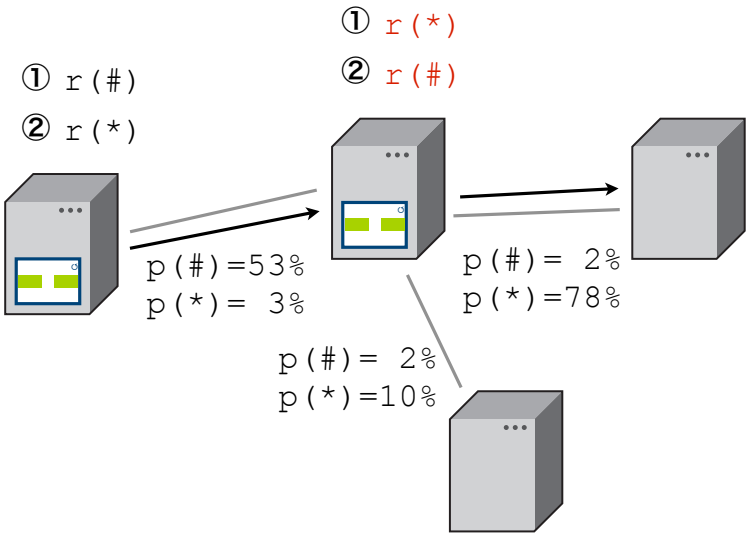


Figure 4.4: Mutable Moving Query Plan – Example

The query is then forwarded to the center node, where also no data is found. However, the routing probabilities for the two keys have now changed. The probability of finding data for $\star$ is now not only higher for all neighbor nodes, it also has the highest overall probability of 78%. Hence, the operator order is changed to read data for key $\star$ first and the query is routed to the node with the highest result probability for this key.

## 4.3 Cost Model – Future Costs and Required Investment

Before we are able to discuss efficiency of the query execution process, we first have define how we define costs in our environment. In distributed databases, efficiency of queries is rated using a cost model, which estimates the resources to be spent when evaluating a single query [Kossmann, 2000]. The cost model therefore defines the goal for query optimization, namely to find the execution plan with the least cost.

As described in [Gounaris et al., 2002], DQP methods typically aim either at minimizing the total response time for a single query or maximizing the overall throughput of the system. In our case, we will focus on single query evaluation, since in order to maximize the overall

throughput, nodes are typically sampled for their current load, which is not feasible in our abstract and generic network model. To minimize the total response time from a distributed system, it is necessary to minimize the total amount of messages sent through the network infrastructure, often also called *hop count*. The reason for this simplification is that network transmissions are costly, typically being orders of magnitude slower than local computation and data access. In comparison, the processing time requirements on the individual nodes are negligible. Hence, the cost model should be focused on minimizing the amount of steps through the network.

However, the size of the messages has also to be taken into account. Since the capacity of network connections is usually limited, the size of the messages sent has a direct impact on the speed at which a message is transmitted. This is especially relevant in our case, where the partial results of the query have to be carried along with the query through the network.

### 4.3.1 Shipping Cost

The size of the partially evaluated plan is unfortunately constantly changing. As new data is fed into the query tree through the read operations, the size of the current plan is increased at first. As higher-order operators are evaluated, its size may be reduced again. Hence, our cost model is based on a notion of *shipping cost*. Shipping costs are only valid for a unchanged state of the query plan, and are incurred on every transition of the query processing operation to another node. Shipping costs are calculated on a per-tuple basis, as the sum of the cardinality of all intermediate results inside the query. This is of course a simplification, as transmitting the query tree also incurs cost, and tuples can greatly vary in the number of key/value mappings they contain. However, the query tree itself is of constant size, and while source tuples may vary in size, they do so equally for all possible plans.

The overall goal of applying the cost model is then to minimize the sum of the shipping costs for all hops taken for the processing of a single query. We use a greedy method to minimize the overall sum of the shipping costs. We express the shipping cost as the sum of all transmissions of a single tuple while processing a single query, and refer to this metric with the term *transmitted tuples*. This metric is a simplification of the total network traffic incurred by each processed query, it can also be compared to the communication complexity [Kushile-

vitz and Nisan, 1997]. Our problem of finding the minimal path through the network can be compared to the Travelling Salesman Problem, for which greedy algorithms have been shown to produce near-optimal results in many cases [Kruskal, J. B., 1956]. Also, the greedy algorithm has the advantage of straightforward execution in a distributed environment.

### 4.3.2 Size and Distance Heuristics

Our cost model reflects this greedy approach: For all equivalent permutations of the query and operator ordering, we calculate the current size of the query plan. We define an abstract heuristic $size$ similar to [Avnur and Hellerstein, 2000; Tian and DeWitt, 2003]. For any routing key $k$, the $size(k)$ heuristic will calculate an estimation of the amount of tuples stored inside the DSS for the routing key $k$. Similar to the routing heuristic presented in Section 3.2, we also model it to be imperfect. The value $p_s$ describes the maximum percentage by which the estimated values may differ from the reality.

Since moving the current query state to the node where this hypothetical future intermediate result may be found also includes some cost, we use a second heuristic $distance$ to determine the distance from the current node to another node in hops for a particular routing key. Similar to $size$, the $distance(n_c, k)$ heuristic calculates the distance in hops from the current position of the query evaluation process $n_c$ to the node where data for a specific routing key $k$ is stored. Consistent with $size$, the value $p_d$ models the maximum percentage by which the resulting values might differ from reality. This again enables us to configure our approach to specific coordination methods.

### 4.3.3 Future Size and Required Investment

Using the $size$ heuristic, we can temporarily set the cardinality of the basic read operator to be evaluated next to this estimated value. This way, we are able to determine which parent operators in the query tree also would be able to evaluate in this hypothetical situation. However, estimating the result set size of these parent operators is difficult. In a traditional database scenario, sampling techniques are used to estimate the size of intermediate results [Harangsri, 1998]. Unfortunately, sampling unread data is not feasible in our scenario, since this would

require locating the data to be sampled first. Hence, only non-sampling techniques are applicable here, for exampling using statistical information such as histograms. However, creating these histograms and making them available to all nodes is also difficult in our distributed system model. For the remainder of this work, we therefore do not estimate costs for parent operators. Using this method, we are nonetheless able to create an indication of the *estimated future size* of the query tree and perform relative comparisons. The algorithm to calculate this metric is given in Algorithm 2.

A possible optimization of this situation would be to maintain a local history of the cardinality of intermediate results for operator subtrees from previous query evaluations and then to use this information for more accurate size estimations. However, since centralized maintenance of these statistics is out of the question in our network model, this collection and maintenance of statistics has to be performed on a per-node basis. A node would analyze partially evaluated queries, in particular already evaluated operators. For these subtrees, local statistics for can be created. A least-recently-used (LRU) cache can be used to limit the memory requirements of these statistics and also minimize cache misses, depending of course on the workload of the DSS. However, we would then face the problem of possible discrepancies between the static cost model based on the heuristics and the locally available statistics. This is also an issue in single-node and centralized databases, here, a feedback loop can be established to allow these discrepancies to be reduced [Markl et al., 2004]. This method is also applicable in our environment.

Using the two heuristics, we can calculate the total costs incurred for reaching the node where a basic read operator could be evaluated. We describe this as the *required investment* in order to obtain the estimated future size. By creating a weighted sum between the estimated future size and the required investment to create it, we are able to rank all permutations of the current query plan. The query optimizer is then able to choose the plan with the smallest cost and switch processing to the new plan. Since the values calculated by the heuristic are allowed to express fluctuations in their results to allow their scalable implementation, we also define a threshold for this switching process, the so-called *minimal improvement*. Only when a new plan is better than the current plan by at least this factor, execution is switched to the new plan.

### 4.3.4 Cost Estimation Example

To show the impact the cost estimation has on the shipping cost, we present an example, where the selection of the wrong execution plan has a huge effect on the processing efficiency. This example scenario is pictured in Fig. 4.5. Here, a query consisting of two basic read operations for keys `a` and `b` as well as a join operator is started on node $S_1$. Node $S_4$ stores tuples containing key `a` with a cardinality of 10 tuples. Node $S_3$ stores tuples containing key `b`, but with a cardinality of 1,000 tuples. We assume, that each tuple from $a$ only has one join partner from $b$, resulting in a result set size of 10 tuples. At the start on node $S_1$, the intermediate size of the query plan is 0 tuples, and according to our cost model, no shipping costs are incurred when this plan without any intermediate results is moved. Two minimal execution plans are possible: The alternatives are either first using `a` as a routing key towards node $S_4$ and then using `b` towards node $S_3$ or the inverse. While producing equivalent results, the shipping costs of these two alternatives vary by two orders of magnitude.

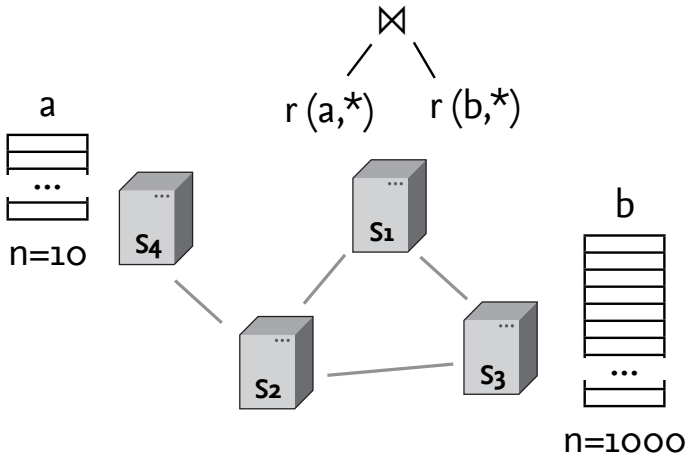


Figure 4.5: Cost Model – Example

The development of the shipping cost is shown in Table 4.1. The left column contains the shipping cost for each transmission of the query evaluation process for the alternative plan, where tuples for `a` is read first, visiting the nodes in the order $(S_1, S_2, S_4, S_2, S_3, S_1)$. This alternative incurs shipping cost of 30 transmitted tuples according to our cost model. In contrast,

| a,b | b,a |
|---|---|
| $S_1 \rightarrow S_2 = 0$ | $S_1 \rightarrow S_3 = 0$ |
| $S_2 \rightarrow S_4 = 0$ | $S_3 \rightarrow S_2 = 1,000$ |
| $S_4 \rightarrow S_2 = 10$ | $S_2 \rightarrow S_4 = 1,000$ |
| $S_2 \rightarrow S_3 = 10$ | $S_4 \rightarrow S_2 = 10$ |
| $S_3 \rightarrow S_1 = 10$ | $S_2 \rightarrow S_1 = 10$ |
| $\sum = 30$ | $\sum = 2,020$ |

Table 4.1: Cost Model – Example Shipping Cost

the plan where tuples for b are read first visits the node in the order $(S_1, S_3, S_2, S_4, S_2, S_1)$. Here, shipping cost of 2,020 transmitted tuples are incurred. We can see how the selection of the first alternative is crucial in this case. Assuming accurate $size$ and $distance$ heuristics, we would have selected the first alternative due to its far lower future cost as determined with a requirement investment of 0 (no intermediate results present) and the greedy strategy that would favor the smaller intermediate result.

## 4.4 Algorithmic Descriptions

In this section, we describe the specific algorithms used to evaluate complex queries in our MMQP concept. We start by introducing the algorithm that uses the cost model presented in Section 4.3 to estimate the future costs likely to be incurred by a specific query plan. From a set of generated result-equivalent query plans, this algorithm is then used in the plan selection process, which selects the plan with the lowest cost. From there, we can then describe the generic query processing algorithm and the generation of equivalent plans. Combined, all these algorithms represent implementations of the optimization and execution phase previously outlined in this chapter.

Formally, a query plan is defined as the combination of a query $q$, which is a reference to the root node of the operator tree as described in Section 3.5, and an ordered list $l_r = (r_1, r_2, \ldots, r_n)$ of read operations to be evaluated next. A query execution plan can then be

given as the 2-tuple $qp = (o_r, l_r)$. The routing key the query operation uses to determine the next node to be visited next is the routing key of the first routing operation in $l_r$. The size heuristic $size$ is defined as a function from a routing key $k$ to a positive integer, such that $s \leftarrow size(k), s \in \mathbb{N}$. While this heuristic has to be calculated locally, the current location of the query execution is not relevant, as the estimated result set size for a particular data identifier is a value constant for the entire DSS. The distance heuristic $distance$ is also a function from a routing key to a positive integer $d \leftarrow distance(n_c, k), d \in \mathbb{N}$. This time, the current location of the query evaluation process is relevant ($n_c$), and thus included in the function signature.

**Algorithm 2** Future Cost Calculation Algorithm for MMQP

**Require:** $size$ heuristic

1: **procedure** FUTUREQUERYSIZE($qp = (o_r, l_r)$)
2: $o_n \leftarrow head(l_r)$
3: $k \leftarrow key(o_n)$
4: $setCardinality(o_n, size(k))$
5: $setEvaluated(o_n)$
6: **return** $treeSize(o_r)$
7: **procedure** TREESIZE($o = (p, c, f, r, e)$)
8: $s \leftarrow |r|$
9: **for all** $co \in c$ **do**
10: $s \leftarrow s + treeSize(co)$
11: **return** $s$

Using these definitions, we can now describe the recursive algorithm to assign any query plan $qp$ with a numeric cost value. The definition of this calculation is given as pseudo code in Algorithm 2. The procedure $futureQuerySize$ retrieves the read operator to be evaluated next ($o_n$) from the operator evaluation list. For the data identifier $k$ assigned to this operator, the estimated result set size is calculated. For this estimated size, the delivery into the operator is simulated, by marking the read operation as evaluated and setting its

cardinality to the estimated size (Line 5). Finally, the estimated future (shipping) cost of this query plan is calculated recursively from the root operator $o_r$ and returned (Line 6).

**Algorithm 3** Query Plan Selection in MMQP

**Require:** $distance$ heuristic, sum weight parameter $weight \in [0, 1]$, minimum improvement $improvement \in [0, 1]$

```
1:  procedure NEXTPLAN(currentPlan = (o_r, l_r))
2:      bestPlan ← currentPlan
3:      QP ← generateQueryPlans(o_r)
4:      currentSize ← queryEvaluationSize(o_r)
5:      lowestCost ← currentSize * (1 − improvement)
6:      for all qp = (o_r, l_r) ∈ QP do
7:          futureSize ← futureQuerySize(copy(qp))
8:          investment ← currentSize * distance(head(l_r))
9:          totalCost ← (investment * (1 − weight)) + (futureSize * weight)
10:         if totalCost < lowestCost then
11:             bestPlan ← qp
12:             lowestCost ← totalCost
13:     return bestPlan
```

This cost model can now be used to determine the best query evaluation plan from the current node's viewpoint. This process is defined in Algorithm 3: For a given query, the set of equivalent query plans is calculated as described in Section 4.4.1. For each alternative, the expected future size of the plan after the evaluation of the next read operator is estimated using the $futureQuerySize$ method (Line 7. The required investment to reach this future size is then calculated using the $distance$ heuristic (Line 8). The total expected cost is then calculated by creating a weighted sum of the two previous values (Line 9). The process is influenced by the parameters $weight$ and $improvement$. The $weight$ parameter determines the influence of the required investment and the future size to total cost, a value of 0.5 describes equal influence and should be the starting point. The $improvement$ parameter describes by how much of a fraction the new plan must be better than the old plan before a

switch between plans in the evaluation process is made. For example, a value of 0.1 requires the new plan to provide at least a 10% improvement over the current plan. If none of the generated alternatives achieve this improvement, the current plan is returned as the best option.

**Algorithm 4** Query Evaluation Process for MMQP

**Require:** Step count limitation $maxSteps$, query plan $qp = (o_r, l_r)$, start node $startNode$

1: $currentNode \leftarrow startNode$
2: $steps \leftarrow maxSteps$
3: **while** $steps > 0$ **do**
4: $(o_r, l_r) \leftarrow nextPlan(o_r)$
5: $op = (p, c, f, r, e) \leftarrow head(l_r)$
6: $T \leftarrow read(key(nextReadOp), value(nextReadOp))$
7: **if** $|T| > 0$ **then**
8: $r \leftarrow T$
9: $f \leftarrow true$
10: $e()$
11: $l_r \leftarrow l_r - op$
12: **if** $evaluated(o_r)$ **then**
13: **return** 'success'
14: $continue()$
15: $currentNode \leftarrow nextHop(currentNode, k, neighbors(currentNode))$
16: $steps \leftarrow steps - 1$
17: **return** 'failure'

Now all basic parts are available to fully describe the algorithm for evaluating MMQP queries. This description is given in Algorithm 4: From a query plan $qp$ including the operator tree $o_r$ and the ordered list of basic read operations to be evaluated $l_r$, query evaluation starts on the node $startNode$, where the query has been received from the application. To ensure termination, the amount of steps inside the network is limited to $maxSteps$. While there are still steps left, the $nextPlan$ function described in Algorithm 3 is executed, which

will generate the best query execution plan by using the three heuristics (Line 4). Then, the local storage (described in Section 3.3.1) is scanned for data matching the basic read operator $r$ with the highest priority (Line 6). If tuples are found, they are added to the read operators result set $r$, the operator its evaluation flag $f$ is set to true as the operator is now fully evaluated, and its evaluation function $e$ is called, which will escalate the results up to its parent operations such as join or selection operators (Line 10). If this evaluation cascade leads to the query being fully evaluated, this can be determined by checking if the root node $o_r$ is fully evaluated. If so, query processing is terminated and the results are sent back to the node the query originated form (Line 12). Furthermore, the basic I/O operation is removed from the list of operators to be evaluated. Finally, the next node to be visited is selected from the neighbor list according to the data identifier of the next basic I/O read operation to be evaluated.

### 4.4.1 Plan Enumeration

Since the very early database "System R" [Selinger et al., 1979], query optimization has been based on two steps: First, enumerate a number of result-equivalent query execution plans, and second, assign costs to them. We have already introduced our cost model in Section 4.3, and will now describe our approach on plan enumeration. In relational database systems, dynamic programming [Bellman, 1957] is typically used to implement a bottom-up enumeration approach: More complex plans are built from simpler sub-plans, starting with the basic I/O operations on the physical tables as smallest unit, and continuing with n-ary join operations. During this time, sub-optimal plans are pruned away constantly. Dynamic programming for query enumeration has been shown to produce the best possible plans if the cost model is accurate [Kossmann, 2000]. However, our error-prone heuristics for cost calculation are not stable enough to allow us to prune plans early.

We have adopted a simple approach based on the widely used concept of rule-based query optimization [Freytag, 1987]. In a nutshell, a set of equivalency rules is defined, which are then applied to an operator tree in order to generate the desired set of permutations. Since the permutations are performed on every node the query evaluation is visiting, special consideration had to be put on *query rewritability*. Rewriteability restricts the operators

that can be subject of permutation for various permutation rules. In our case, operators already evaluated may not be reordered at all. Algorithm 5 defines our approach to query and operator evaluation order permutation. Quite simply, the set of permutation rules is applied until no rule is applicable any more.

**Algorithm 5** Query Plan Permutation

**Require:** query plan $qp = (o_r, l_r)$, set of permutation rules $R$

1: $P \leftarrow \{qp\}$
2: $rf \leftarrow true$
3: **while** $rf$ **do**
4:  **for all** $r \in R$ **do**
5:  $rf \leftarrow false$
6:  **for all** $o \in listOperators(o_r)$ **do**
7:  $qp_n \leftarrow apply(r, o)$
8:  **if** $qp_n \notin P$ **then**
9:  $rf \leftarrow true$
10:  $P \leftarrow P + qp_n$
11: **return** $P$

While being not particularly efficient, the amount of CPU usage on each node is not our main focus in this work. The improvement of the efficiency of the enumeration approach presented here is therefore deferred to further work. To give an example for a transformation rule, let us consider join swapping: If a join operation has a child operation that is also a join operation, the parent join operation may be replaced with the child operation, with the children of both operators being correctly reattached. An application of this rule is pictured in Fig. 4.6, where the join operations $\bowtie'_a$ and $\bowtie'_b$ change their places. Whether this reorganization is beneficial depends entirely on the size distribution between those operators, which again is estimated by the cost model.

Due to the close relatedness of our query model, we were able to re-use the equivalency rules from relational databases, e.g. from [Codd, 1990]. Table 4.2 lists the equivalency rules that we have also found to be compatible with our model.

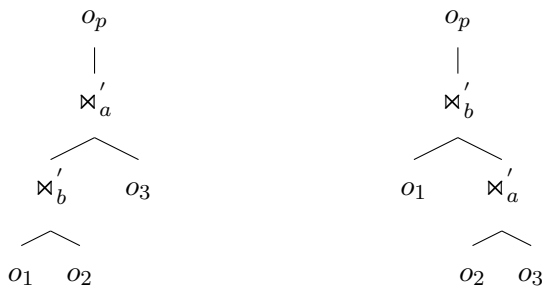


Figure 4.6: Permutation Rule – Join Swapping

| | | | |
|---|---|---|---|
| $\sigma'_{a_1\theta_1p_1,a_2\theta_2p_2}(T)$ | $\equiv$ | $\sigma'_{a_1\theta_1p_1}(\sigma'_{a_2\theta_2p_2}(T))$ | Conjunctive Selections |
| $\sigma'_{a_1\theta_1p_1}(\sigma'_{a_2\theta_2p_2}(T))$ | $\equiv$ | $\sigma'_{a_2\theta_2p_2}(\sigma'_{a_1\theta_1p_1}(T))$ | Commutative Selections |
| $\pi'_{a_1,a_2,\ldots,a_n}(\pi'_{b_1,b_2,\ldots,b_m}(T))$ | $\equiv$ | $\pi'_{a_1,a_2,\ldots,a_n}(T)$ | Superfluous Projections |
| $T_1 \bowtie' T_2$ | $\equiv$ | $T_2 \bowtie' T_1$ | Commutative Joins |
| $T_1 \bowtie' (T_2 \bowtie' T_3)$ | $\equiv$ | $(T_1 \bowtie' T_2) \bowtie' T_3$ | Associative Natural Joins |

Table 4.2: Plan Enumeration – Operator Equivalency Rules

A very important transformation rule is changing the order in which the basic read operators are to be executed. While their evaluation order does not change the operator structure further up in the query tree, their order of evaluation does have a profound impact on the incurred costs, as we have showed above.

## 4.5 Failure Recovery

When determining the list of assumptions and preconditions for MMQP in Section 4.1, we have specifically allowed routing errors consistent with our overall network model. Furthermore, we considered the possibility of nodes or network connections failing during query processing. Also, the location of data was not fixed in the network at any time. An additional situation not previously discussed is the node the client application connected to and started a query on. Since the client application also expects results being delivered

to them from this very node, the failure of this so-called “entry node” has even higher implications for MMQP. In this section, we discuss the possible errors as well as means of their detection and methods to make them transparent to outside applications.

### 4.5.1 Misrouted Operations

For our very general network model presented in Section 3.1, we have already shown that allowing routing errors decreases the coherence in the network, but allows additional scalability. The case of single retrieval operations alone being not forwarded correctly is therefore not the exception, but the norm. While we have shown in Section 3.2.3 that single read operations are very likely to eventually encounter the searched data items, there is still a rather extensive margin for error. Of course, since MMQP also uses this routing process to find data matching the leaf operators in the query tree, it is also prone to being misrouted.

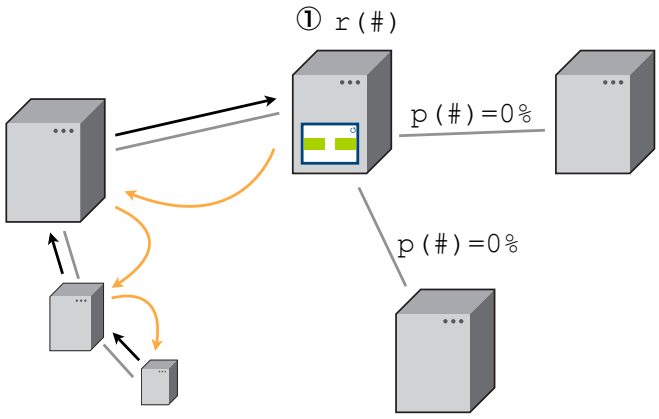


Figure 4.7: Mutable Moving Query Plans – Routing Failure Recovery

Consider the situation depicted in Fig. 4.7. Here, the query evaluation process is on a node trying to find data for a key, e.g. #. However, from evaluating the routing function, we have found no indications where to go next in search of data matching #. It is therefore likely that either no data is available for #, or that the query evaluation process has been routed to an area of the network where on the one hand no data is available for the key, and on the other hand no routing information for this key is available. Bear in mind, that this situation is also

possible in a network with a routing function enjoying a very low error probability as per our network model, since this probability describes the routing performance of all nodes.

In this case, we introduce the following process: The MMQP process carries a list of node identifiers it has visited on its path through the network in search of evaluating the current query. Whenever the routing information is below a certain threshold, the process starts tracking back its path through the network, in search of the node where the "wrong turn" was taken. This point can be identified again by the routing probabilities. If we find a node where two or more neighbors have non-zero routing probabilities for the key the process is currently searching data for, we select the neighbor node not previously on our stored path. This will allow to escaping these local minima, without having to track back to the node the query evaluation process originated on. Of course, these backtracking steps count in the current processes step count, which will terminate the process once it has reached its step count limit. Therefore, the backtracking will not lead to unbounded activity.

As mentioned in Section 3.2.1, the basic retrieval operation that operates on our abstract network is unable to determine that data for a specific key does not exist within the storage network without trying to locate the data and then being terminated by the step count limit. Since the query processing operation is in essence a sequence of several basic retrieval operations, it exhibits the same behaviour. Therefore, queries that contain a key for which there is no data will commence backtracking soon. However, in this case backtracking will be fruitless, as there is no place where the routing probabilities for the non-existing key will be higher than the potential noise generated through the routing error probability. The process will hit its step count limit and be terminated. Therefore, there is no conceptual difference between queries that are mis-routed and queries that are looking for non-existing data.

### 4.5.2 Node or Network Failure

As the DSS increases in size, the probability of any node being unable to perform its function or its network connections to fail increases steadily. These failures can occur at any time, but are particularly troublesome when this happens during the process of evaluating a single query. We only consider these inter-query failures here, since they directly impact our

query processing method. Due to the lack of central control, only the node the query is currently processed on is aware of its current location. Therefore, if this node fails, the query processing will silently fail, too. Equally, if a query is being forwarded to a neighboring node, a network failure on this connection will stop the query evaluation process. We therefore have to differentiate those two cases:

First, if a partially evaluated query plan is forwarded to a neighbor node, this transmission may fail. This condition is easily detected in many cases, for example if a network protocol that acknowledges received packets is used. In this case, the sending node can select another neighbor node as the recipient of the query plan currently being processed.

Second, a node may fail after receiving the query from a neighbor node, and before it is able to forward it again. Other nodes could detect this condition. There are a number of solutions that have been proposed to detect failure, from central heartbeat systems over gossiping approaches to de-centralized probabilistic methods [Aguilera et al., 1997; Gupta et al., 2001]. Unfortunately, even if neighboring nodes are able to detect the failure of a node, they would still be unable to salvage the query that was currently running on this node. In order to achieve this, the entire process would have to be duplicated, which is not desirable due to its performance penalties.

However, the node where the client application has connected to ("entry node") and where the query evaluation process was started is an ideal candidate for tracking the query, since the client application is also expecting results from there. The entry node is also able to keep the query sent by the application until results arrive, and is thus can restart a query, if no results arrive. Unfortunately, the only means for the entry node to determine whether the query has been lost is through introducing a time limit for query processing. For each query started on this node, a timer is started. When the timer reaches the time limit, the query can be restarted. Since queries may be started on any node, this does not introduce a central point of failure and also allows us to detect failures of the mentioned nature. However, the entry node may also fail. In this case, the connection to the waiting client application will also be terminated, a contingency it has to be prepared for. Therefore, we are able to provide run-time level fault tolerance in general, where these failures are transparent to the application. However, if the entry node fails or is disconnected, only application-level fault tolerance is feasible.

## 4.6 Abstraction and Efficiency Analysis

While the previous part of this section has been focused on distributed query processing, we now describe the an abstraction of our approach on distributed query processing. This serves two purposes: First, by abstracting from DQP, we can identify other application fields for our approach. Second, this abstraction simplifies a stochastic efficiency discussion similar to Section 3.2.3. In presenting MMQP, we have already described how query evaluation can be compared to a journey through the DSS under constraints, which was also proposed in [Bharath-Kumar and Jaffe, 1983]. We have already argued for our greedy approach to perform this journey with near-optimal costs.

The notion of the *required investment* and *estimated future size* or *gain* was used to prioritize the evaluation basic read operators, that define where the query plan needs to be routed next. Abstracting from the query plan, we now consider the abstract form of Multi-Objective Processes (MOP) within a DSS. While not necessarily focused on retrieval, all that is required from these processes in order to be optimized by our approach is the exhibition of an ordered list of data identifiers or routing keys to be visited. We also assume that the process has to be routed to all operators in this list in order to finish its task.

Reconsider the algorithms previously presented both to select the best query plan and to execute said plan: All that is required to adapt this process from DQP to generic multi-objective processes are redefinitions of the way the plan permutations are created, how costs are calculated, how results are returned, and how success is determined. We can therefore adapt said algorithms in a generic way. We define a MOP as a 4-tuple $MOP = (l_r, d(T), f, s(k, n))$. As before, $l_r$ is a set of basic read operators, each looking for data matching a single data identifier. $d(T)$ is the delivery function, called as soon as data looked for by one of the read operators is found. $f$ is the finish flag, which is changed by the process as soon as it considers itself completed. Through the use of the heuristics $nextHop$, $size$ and $distance$, we can perform our greedy algorithm and select the next read operator to be evaluated from $l_r$. To execute our *Investment/Gain* approach, the MOP also includes the simulation function $s(k, n)$. This function estimates the impact on the size of the process,

if $n$ tuples matching the identifier $k$ are found. This is sufficient to calculate investment (through the $distance$ heuristic and the current size of the process.

### 4.6.1 Stochastic Analysis

While the multi-objective process abstraction from the previous section extends the applicability of our approach, we now turn to the stochastic analysis. Similarly to the analysis we have already performed for our coordination method in Section 3.2.3, we now investigate the theoretical performance of our approach with regards to the size of the network. The method by which we perform this analysis is a thought experiment. Since there are many variables that influence the shipping cost, we have to restrict our considerations to only show the influence of the network size. We assume a fixed dataset and a fixed query, that is evaluated over a random network. Also, we only discuss the average case, which allows us to re-use the results from the single-element retrieval analysis.

Given a network $G = (N, L)$ and a $MOP = (l_r, d(T), f, s(k, n))$, we can assume that data matching the read operators in $l_r$ is distributed over various nodes from $N$. We assume the $size$ heuristic to be perfectly accurate, thereby yielding the exact cardinality for each routing key. We denote this artificial heuristic by $size_a(k)$. The intermediate size of the $MOP$ is changing every time data is added on one of the nodes with matching data. For a path through the network $p = (n_1, n_2, \ldots, n_m)$, we model the intermediate size as a function $is(n) \rightarrow \mathbb{N}$ in transmitted tuples. Since we assume the network connections to be unweighted, the total shipping cost of evaluating the process is $\sum_{j=1}^{m} is(j)$ transmitted tuples. As we do not know the future intermediate size in advance, it is estimated analogous to our cost model using the $size$ and $distance$ heuristic as described in Section 4.3.

However, given a static data set and an unchanged read operator evaluation order $l_r$, the absolute values of the $is$ function are static, since the cardinality of the delivered base data as well as the internal processing (such as a query plan) also remain static. Therefore, the total cost only depends on the distance this intermediate result has to be shipped through the network. In our analysis of the basic retrieval operation in 3.2.3, we have determined the average amount of hops to be expected when moving from one node to another node in the network in the presence of error-prone routing. This relationship between network size

and average path length was shown to be logarithmic. Visiting multiple locations therefore also has this logarithmic complexity, and increasing the network size will – on average – also have a logarithmic effect in the shipping cost, which is a crucial precondition to the scalability of our approach. We test this prediction in an experiment in Section 5.4.4.

Furthermore, as stated above, we are using a greedy algorithm, that tries to minimize the cost for the immediately following step that is taken. Therefore, we are also interested in the performance penalties of a greedy algorithm. The class of algorithms that our approach can best be compared to are those that aim to solve the Travelling Salesman Problem (TSP). A fitting greedy-style heuristic for TSP is the Nearest-Neighbor (NN) algorithm. This approach is the only one of several greedy methods that computes the solution in one pass starting from a random node, which makes it comparable to our approach. NN mimics a traveler whose rule of thumb is always to go to closest un-visited location. It constructs a path $\pi$ through the network, which orders the visited nodes $n_{\pi(1)}, n_{\pi(2)}, \ldots, n_{\pi(N)}$. In general, the next node $n_{\pi(i+1)}$ is chosen such that the distance $d(c_{\pi(i)}, n_{\pi(i+1)})$ is minimal [Johnson and McGeoch, 1997].

| $\lvert N\rvert =$ | $10^2$ | $10^{2.5}$ | $10^3$ | $10^{3.5}$ | $10^4$ | $10^{4.5}$ |
|---|---|---|---|---|---|---|
| NN | 1.3 | 1.8 | 2.4 | 3 | 3.6 | 4.1 |

Table 4.3: TSP – Nearest Neighbor Overhead [Johnson and McGeoch, 1997]

In experiments running NN on random networks with random distances, NN was found to have a considerable overhead in terms of path length over the optimal round-trip according to [Held and Karp, 1970]. These overheads are reproduced in Table 4.3. For example, for a network of 1,000 nodes, NN required 2.4 times more hops than the theoretical lower bound. While our Investment/Gain approach uses a slightly more complex metric to determine costs, an overhead similar to that of the NN approach has to be expected. We will again compare these predictions with our experimental results in the following chapter.

## 4.7 Summary and Conclusions

In this chapter, we presented our approach at distributed complex query processing, the Mutable Moving Query Plans. Here, the execution plan of the query is continuously optimized while being sent on a journey through the DSS and while collecting partial results. For this approach, we have presented an abstract description, an architecture for evaluation, a cost model based on a notion of investment and gain, and specific algorithms for query optimization, query execution and routing failure handling.

Also, we have abstracted our approach away from query processing and showed a generic model of Multi-Objective Processes inside a DSS, which potentially enables our approach to be applied to many other challenges besides query processing. Furthermore, we have based our theoretical stochastic analysis on this model, thereby not only showing the scalability of our generic concept, but also of our approach to query processing.

Using the taxonomy on adaptive query processing presented by [Gounaris et al., 2002], we are now able to classify the query processing approach presented here as follows: Both *reformulation* as well as *operator reordering* of the remainder of the query plan are used. The focus of the methods is adapt the point in time for *data arrival* at the node that started the request, they are thus aimed at *minimizing the total response time*. Responsibility for optimizations is *local*, where each node can decide whether an adaption is required. Finally, the environment in which the adaption happens is a highly *distributed* storage system.

We have presented a method to perform complex query processing on structured data within a DSS using our abstract network model here. While we have predicted a logarithmic behaviour with regards to the size of the DSS from our stochastic discussion, these predictions will be put to the test in the following chapter. Since we were able to describe a method for distributed query processing solely based on the building blocks defined in the previous chapter, we can also regard these building blocks to be theoretically sufficient for both effective and efficient query processing.

# 5 Verification Methodology and Experiments

In the previous chapters, we have presented a method for complex query processing on an abstract network model. We have made several predictions on the performance of both our model, its basic read operation, and our MMQP method. However, while we have presented theoretical and stochastic discussions of these components, we have to confirm our predictions through experiments. In this chapter, we perform these experiments: First, whether the distributed query processing approach presented is effective in processing queries. In this context, effectiveness is the ability to produce correct results at all. Second, we investigate the efficiency of the proposed approach with regards to our cost model, which is based on shipping cost. We are particularly interested on the impact of the properties of the network model on these costs.

We start this chapter by describing our experimental methodology, which is based on controlled experiments on a simulated DSS. We then shortly introduce our test environment, in particular the synthetic heuristics used in this environment. Also, our test data set and test queries based on the TPC-H benchmark are described [TPC, 2011].

Our road map for our experiments builds aims at collecting evidence supporting our hypotheses and predictions in a cumulative fashion: We start by verifying the behavior of our simulation testbed network structure, since all subsequent processes use this model and are dependent on its designed characteristics. In particular, the average hops required for single-element retrieval in various network sizes are tested.

Subsequently, we test complex query processing by first testing the effectiveness of query processing, specifically that our query evaluation approach produces the correct results for

the test queries. Then, we test whether the two main components – continuous optimization and movement – are effective in reducing the total cost of query evaluation. Once the effectiveness of our components and overall process has been tested, we continue with the impact of environment influences such as the size of the network and the routing error probability as well as the impact of internal parameters to the total query evaluation costs.

## 5.1 Verification Methodology

As we have seen in the previous chapters, a distributed storage system with support for complex query processing is an environment of high complexity and a plethora of different variables, which all potentially contribute to its behavior. On the network level, the size and structure of the network are arbitrary and the heuristics we use to route request and predict distance and size are designed with probabilistic behavior. Furthermore, the structured data to be stored is unknown a-priori, as well as the distribution and degree of locality of the data in the network. Also, the queries being asked on the stored data are also unknown, even if they are composed of a small set of operators, their tree structure introduces another degree of complexity. Also, multiple configuration parameters are present in our query processing approach, most notably the hop count limit in basic retrieval and the weighting factors in query plan selection. Due to its randomized nature, verifying effective and efficient functionality in this system cannot be performed using analytical methods. For example, *process calculi* such as CCS and $\pi$-calculus are based on deterministic behavior of the analyzed processes [Philippou and Michael, 2006], which we are unable to guarantee. On the other hand, techniques such as *probabilistic automata* are essentially Markov Decision Processes, for which also only stochastic properties can be checked [Segala, 2001].

While this non-provability represents a drawback of our approach, we maintain that its adaptability to many specific and provable situations outweighs this issue. The methodology of choice to gather support for our assumptions is therefore the method of controlled experiments. In controlled experiments, we are able to control all variables but one, the independent variable. By repeating the experiments several times with different settings for the independent variable, we are able to give an indication of its effect to the overall process.

However, a second difficulty arises: Since the used heuristics are designed with probabilistic behavior, a single controlled experiment is not suitable to show the effect of the controlled variable. For the subsequent experiments, two main requirements therefore arise: We have to ensure experimental validity, and also statistical significance of results where the effects of heuristics are prevalent.

Experimental validity is divided into *internal validity* and *external validity* [Shadish et al., 2002]. Internal validity refers to whether the independent variable has a measurable effect on the experimental results. Main threats to internal validity include changing states in the experiment subjects. However, in our case, where the entire experiment is a simulation, we can create "clean" preconditions for every test, thereby avoiding these threats. However, we will still have to test for internal validity, in our case for example by using statistical methods. External validity is the process of generalizing from the experimental results to circumstances not covered by the experiment. While repeating the experiment in all possible circumstances with the same observations would be a strong indicator for external validity, the complexity of the observed system inhibits that. Rather, as proposed by [Tichy, 1998], we have chosen to use a set of benchmark scenarios, in which the experiments are performed. The results from the experiments in the benchmark scenario can then be used to indicate external validity, provided the benchmark captures a relevant subset of all possible scenarios.

However, results from empirical research by way of experiments do not prove that our proposed methods will exhibit an observed and predicted behavior in every case. This is particularly true in the light of the huge parameter space, of which any experiments can only capture a small fraction. If observations from experiments should concur with our predictions, in theory this only proves that we were unable to contradict our assumptions with those experiments, and does not allow us to assume their validity [Popper, 1959].

## 5.2 Test Environment

To perform controlled experiments, we have implemented a simulation environment to test the behavior of our models and algorithms and to collect measurements that support or refute or predictions. In this environment, an arbitrary number of virtual network nodes can be

created. Each virtual node maintains a list of virtual network connections to other nodes, consistent with our network model introduced in Section 3.1 Also, each node maintains locally stored data, and provides the local storage operations to read and write tuples locally as described in Section 3.3.1.

### 5.2.1 Routing Heuristic

Since our network model was intended to be abstract, the issue was now how to maintain this abstraction in our simulation environment. In particular, the abstract routing method we have presented was based on assigning scores to connections to neighbor nodes according to the probability of finding data for a specific routing key by following the respective connection. Furthermore, the routing method was deliberately allowed to produce wrong answers according to a error probability $p_f$. We have solved this issue by keeping track of the location of all data and corresponding routing keys in the network at a central location.

Calculating the routing probabilities for any key on any node was now performed by first reading the actual location of the data for the current routing key from this central data structure. Then, we determined the shortest path $R$ through the network from the current node to the node where the data is located on using Dijkstra's algorithm. The neighbor node on this path is now obviously the best choice to find data matching the routing key. To model the error probability $p_f$, we allowed this value to be set in the simulation configuration and used it to add noise to the previously generated result. Formally, for all nodes $n$ in the neighborhood $H$ of the current node, these probabilities were calculated as follows:

$$p(n \in H) = \begin{cases} 1 - p_f & n \in R \\ \frac{p_f}{|H|-1} & otherwise \end{cases}$$

### 5.2.2 Distance and Size Heuristic

The $distance$ and $size$ statistics from our cost model described in Section 4.3 are created similarly. First, the correct value is determined from the simulated network, and then some noise is being added to the result. Contrary to the routing heuristic, the distance and size heuristics do not provide a distribution, but a single value for each input. Therefore, we

overlay a configurable percentage of random noise over the correct result determined from the simulated network.

To obtain the value of the $distance(n_c, k)$ heuristic, we again determine the node $n_d$ where data for key $k$ is stored. Then, we measure the length of the shortest path between the current node $n_c$. This result is then modified with noise $p_d$ such that the result distance $d$ varies randomly in the range $[\lfloor d - d * p_d, \rfloor, \lceil d + d * p_d \rceil]$.

For the $size(k)$ heuristic, we also determine the node where data matching $k$ is stored, and then count the number of tuples stored on that node that match $k$. This result is also modified with noise $p_s$ such that the result size $s$ also varies randomly in the range $[\lfloor s - s * p_s, \rfloor, \lceil s + s * p_s \rceil]$.

In the interest of experiment repeatability we have not considered collecting node-local statistics to potentially improve the accuracy of these heuristics in the following.

### 5.2.3 Data Set and Test Queries

We have selected queries and data from the TPC Benchmark H (TPC-H) (Version 2.14.3[1]), which is aimed at testing decision support systems that examine large volumes of data with queries having a high degree of complexity. This fits to our approach of a large-scale distributed storage system, which may not be able to provide real-time transaction processing due to a relaxed coordination model and thus lack of routing efficiency, but could still be used in a data warehouse scenario, where queries are ad-hoc, but have less rigid timing constraints. TPC-H consists of a data generator for relational data according to a manufacturing use case as well as a set of complex queries. The relational schema contains information about orders. Each order contains a list of items that are part of the order. Each item is supplied by a particular part from a particular supplier. Each order is also assigned to a specific customer. Suppliers and customers are in a particular nation, which in turn is part of a region. The scenario is particularly suited to show the advantages of selective data access, since a relatively small schema here contains relations with huge amounts of tuples, which would otherwise have to be retrieved in full. Appendix A.1 reproduces the full relational schema.

[1] `http://www.tpc.org/tpch/spec/tpch2.14.3.pdf`, accessible as of 2012-07-29

Using the the TPC-H data generator `dbgen` with a data size scaling factor of 0.01, we have generated a set of 86,805 tuples to be used in all experiments. The generated tuples were translated into our schema-less data model according to the translation method in Section 3.3, where the relation name is added to each tuple and the other tuple entry keys are prefixed with the relation name in the case of collisions. For each test run in the experiments, this data set was loaded into our simulation environment of varying size according to the data distribution scheme described in Section 3.4.

Not all of the 22 queries defined in the benchmark could be used for our experiments due to unsupported operators. Queries Q3, Q5 and Q10 were however found suitable, since they only contained conjunctive selections, and no secondary selections. Also, grouping and sorting of the results only occurred at the outermost level for these queries, e.g. the query tree root. We have made made these queries compatible with the SPJ model presented in Section 3.5 by implementing order ($O$) and group ($G$) operators.

In short, Q3 determines orders with high shipping priority, by selecting orders that have not been shipped yet, but have high potential revenue. Q5 determines the total revenue for orders where both supplier and customers are in the same nation, by nation. Q10 finds the top 20 customers, where the lost revenue was greatest due to the customer returning parts. These queries were transformed from their original SQL representation into our query model. The SQL representations and their transformation into our modified SPJ model are given in Appendix A.1.

## 5.3 Single-Element Retrieval

We have described our coordination model based on probabilistic routing in Section 3.2. Here, we validate our stochastic predictions for the relationship between the number of nodes, the amount of neighbors each node has, and the failure probability for the routing function. The accuracy of our predictions is a crucial precondition for the later experiments on our query processing approach, since MMQP in essence combines multiple single-element retrieval operations.

Inside the simulated storage network, we placed a single data item on a random node, and then started the retrieval algorithm searching for this data item from another random node. Using our synthetic routing heuristic described above, we have then executed Algorithm 1, our generic retrieval algorithm on the simulated network structure. Starting from any node in the network, this process calculates the routing probabilities for a given key at the current position, chooses the best suited neighbor node, and continues there. This process repeats itself until data for the carried key is found.

In our stochastic analysis presented in Section 3.1, we have predicted that the network size, the average degree between nodes, and the routing failure probability $p_f$ contribute to the number of hops that have to be performed between nodes in order to reach a specific node or retrieve a particular data item. The hypothesis tested in this experiment is that we are able to correctly predict the average number of hops required to retrieve a data item for heuristics-based request routing. We have measured the amount of hops the retrieval process required by our retrieval algorithm to find the data item. To ensure statistical soundness, the retrieval process was repeated 100 times for each network configuration and the amount of hops averaged. To test the influence of the three parameters, we have repeated this experiment with all permutations of the following settings:

- Network Size: 100, 250, 500, 750, 1000
- Neighbor Limit: 5, 10, 50, 100
- $p_f$ Settings: 0.7, 0.5, 0.25, 0.1, 0.01

In total, 11,000 single retrieval operations were performed in this test. For sake of simplicity, we start by showing the results of our experiment for a neighbor limit of 10 in Fig. 5.1: Here, the averaged amount of hops required is plotted against the different network sizes. For each $p_f$ setting, a different line is plotted. We can see how the $p_f$ parameter directly influences the retrieval performance. The experiment with a routing error probability of 70% (□ symbol) displayed the worst performance, requiring on average 20 hops to retrieve the data item in the network with 1,000 nodes. $p_f$ settings below 50% (• symbols) scored much better, confirming our predictions from Section 3.2.3.

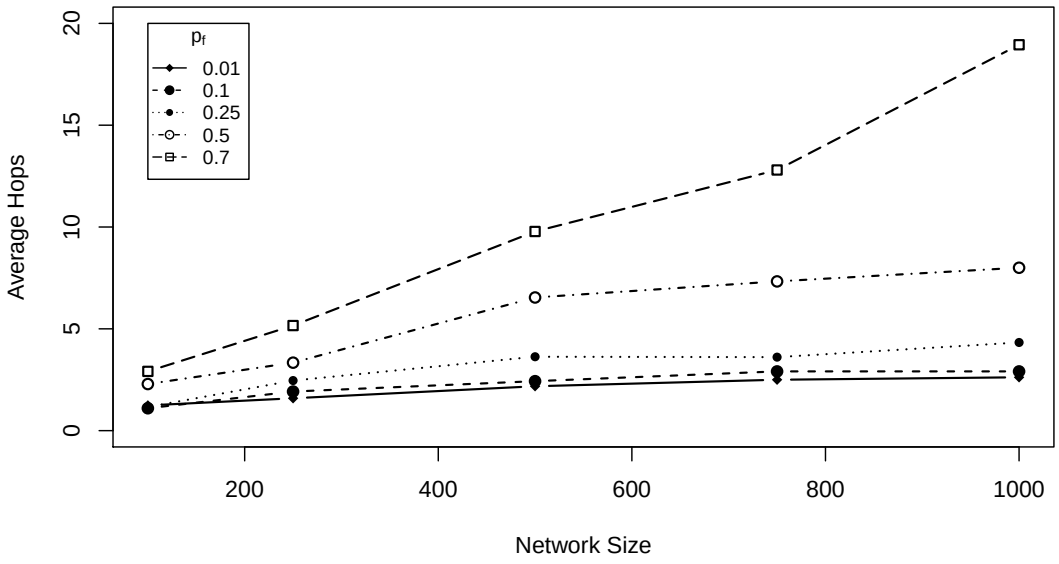


Figure 5.1: Probabilistic Routing – Experimental Results – 10 Neighbors

We have also calculated and recorded the average path length in our generated networks. As we have seen in Section 3.2.3, the average path length in the most general case of random networks is dependent only on the amount of nodes and their average connectivity. Since the expectation for the amount of hops is also directly dependent on the average path length, this enables us to compare all results without having to consider the network structure. The results of this comparison are given in Fig. 5.2: The average amount of hops required to retrieve a data item is plotted against the average path length of the network this retrieval process was performed in. Each $p_f$ setting again is shown on its own line to show the influence of this parameter. Again, this parameter had great influence on the performance of the retrieval process: Settings below 50% (three bottom lines) performed much better than the higher settings of 50% and 70%, again confirming our previous expectations. From the two experiments, where both the network size as well as the dependent average path length were the independent variable, we can assume that our simulation network exhibits the predicted behavior and is thus suited for the subsequent experiments.

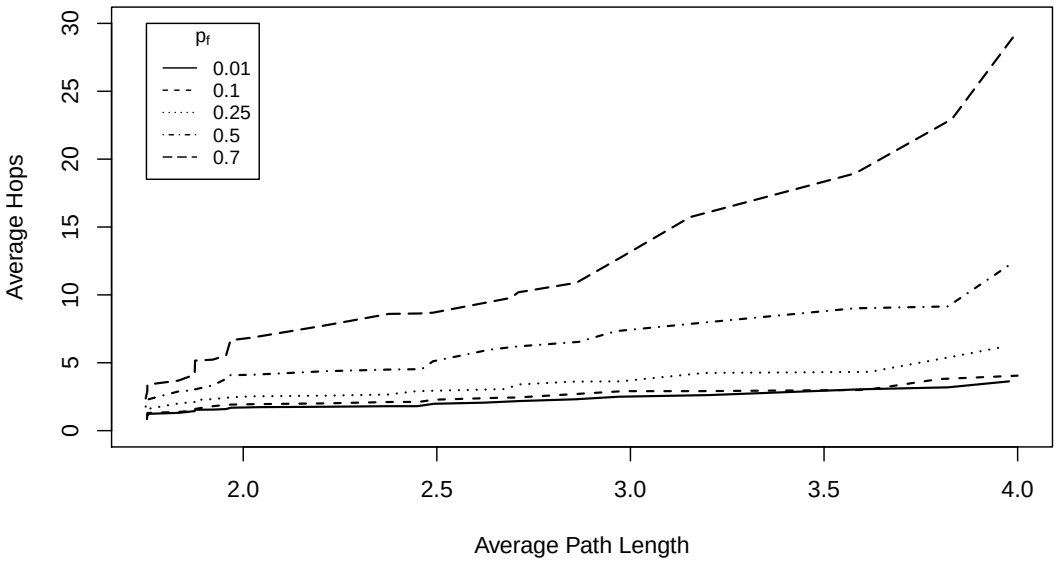


Figure 5.2: Probabilistic Routing – Experimental Results – Average Path Length

To test the relationship between the number of hops predicted by our stochastic model based on the average path length and the $p_f$ setting, we compare those predictions and the experimental results in Fig. 5.3: For each $p_f$ setting, a separate graph is plotted. Each graph contains three sets of data points for hop count according to average path length. The dotted line represents the linear prediction calculated using our stochastic model. The solid line shows the average hops required to retrieve a data item as already shown before. The dashed line shows the maximum number of steps that required for retrieval in our experiment. We can see how prediction and experimental average correlate closely for all $p_f$ settings, which validates our synthetic configurable routing heuristic presented above. Furthermore, we can observe considerably higher worst-case performance, as the failure probability increases.

While the graphical results already show a correspondence between predictions and measurements, we have also calculated two statistical measures between the predicted and the measured average number of steps: Table 5.1 shows both the Pearson product-moment

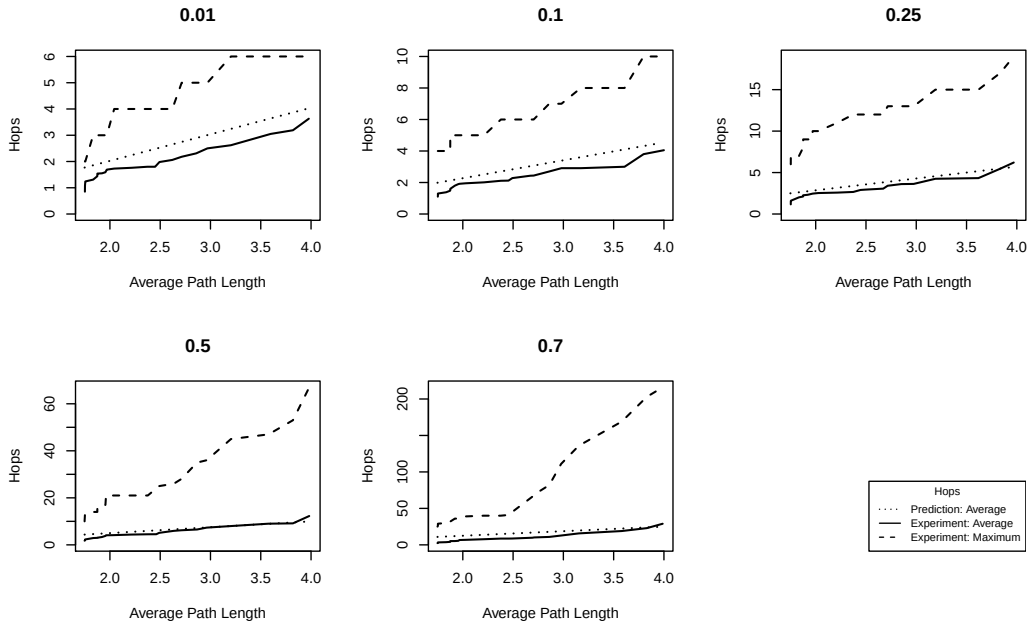


Figure 5.3: Probabilistic Routing – Predictions and Results

correlation coefficient as well as the result of the Lack-of-Fit (LOF) test [Neter et al., 1985] for the different routing failure probability configurations.

The Pearson correlation coefficient describes the covariance of two variables divided by the product of their standard deviations. The coefficient is common to show the linear dependence between two variables, in this case the predicted and measured hop count. However, since this test does only test correlation and not similar values, we additionally performed the LOF test, which in essence adds the quadratic distance between two corresponding data points, and then maps it to a $\mathcal{F}$ distribution.

The closeness of the correlation results to the maximum value of one indicates a strong correlation between the predicted and measured results. Furthermore, if the values for the LOF test are interpreted as F-Test results, we can see a less than 10% probability of the prediction not correctly modeling the observed values.

| $p_f$ | Correlation | Lack-of-Fit |
|---|---|---|
| 0.01 | 0.98 | 4% |
| 0.1 | 0.97 | 6% |
| 0.25 | 0.97 | 7% |
| 0.5 | 0.98 | 4% |
| 0.7 | 0.97 | 5% |

Table 5.1: Probabilistic Routing – Prediction and Experimental Results

From these results, we can make three observations: First, the amount of hops to retrieve a data item is indeed linear to the average path length inside the network. Second, as this metric is fixed in a static network, the most influential parameter for our retrieval algorithm is the routing failure probability $p_f$. Third, our synthetic routing heuristic closely follows the stochastic prediction for the average number of hops required, making it fit for use in the subsequent experiments.

## 5.4 Complex Query Processing with MMQP

While we were able to control most relevant variables in the previous experiment, this is no longer possible for complex queries. Network environment, configuration variables, data and queries all potentially contribute to experimental results. As already mentioned, we have therefore used a benchmark for our distributed query processing approach [Tichy, 1998]. Since we are using simulations, we are able to closely observe the query evaluation process and analyze these observations. However, the effects of randomness inherent in our heuristics will also be visible here. Therefore, repetition and statistical aggregation will be used to remove these effects. We will first test query evaluation effectiveness and then determine whether our predictions regarding evaluation efficiency are accurate.

### 5.4.1 Query Evaluation Effectiveness

Before we are able to experiment on the efficiency of our query processing approach, we must first ascertain whether it is effective, in particular whether it produces correct results. Since we have designed our query model, operators and execution algorithm according to industry standards, our hypothesis and prediction is that our implementation produces the correct results.

The purpose of this experiment was to ensure the effectiveness and correctness of our distributed query processing approach with the TPC-H test data and test queries. To measure this, we have loaded the test data set both into our network simulator as well as the off-the-shelf relational database HSQLDB[2]. Then, we have run the complex test queries on both HSQLDB and our MMQP simulator and then compared the results. We have recorded the result sets for each query on the two systems. We can assume effectiveness of our approach if the results are equivalent. Table 5.2 shows a comparison between the result set sizes for the queries and the two systems.

| Query | HSQLDB | MMQP |
|---|---|---|
| Q3 | 138 | 138 |
| Q5 | 5 | 5 |
| Q10 | 399 | 399 |

Table 5.2: Query Evaluation Effectiveness – HSQLDB and MMQP

We can observe that both systems produced result sets of equivalent size. However, result set size is not sufficient to ascertain equivalent results. Therefore, we have also compared the result sets. Table 5.3 shows an example results for Query 5. To reiterate, Q5 determines the total revenue for orders where both supplier and customers are in the same nation. We can see how the result sets are equivalent, which was also the case for all three queries. We therefore assume that our simulation of MMQP is effective in evaluating the test queries, since the results were equivalent with those created by a commercial database.

[2] `http://www.hsqldb.org`, accessible as of 2012-07-29

(a) HSQLDB

| n_name | revenue |
|---|---|
| VIETNAM | 1000926.7 |
| CHINA | 740210.76 |
| JAPAN | 660651.24 |
| INDONESIA | 566379.53 |
| INDIA | 422874.68 |

(b) MMQP

| n_name | revenue |
|---|---|
| VIETNAM | 1000926.7 |
| CHINA | 740210.76 |
| JAPAN | 660651.24 |
| INDONESIA | 566379.53 |
| INDIA | 422874.68 |

Table 5.3: Execution Effectiveness – Example Results for Query 5

### 5.4.2 Component Effectiveness

Our distributed query evaluation approach presented in Section 4.4 relied on two main concepts: First, we assumed that moving the evaluation process through the network as well as constantly reordering the query evaluation plan would increase overall efficiency with regards to the shipping cost.

To verify this assumption, this experiment compares the hop count and the shipping cost of the evaluation process, which we have chosen as the basis for our cost model in Section 4.3. To test the influence of the "Movement" and "Reordering" components of our concept, we have extended our implementation to allow them to be enabled and disabled on demand. If movement is disabled, the query evaluation process returns to the node it originated from after each single-key retrieval operation. If reordering is disabled, the query is evaluated "as-is", and no optimization at all takes place. Furthermore, this experiment is aimed to show a relative difference, the various parameters were fixed to common sense values. Our prediction for the results is that both movement and reordering improve the overall efficiency of query processing.

As a testbed, we have created a simulated network containing 1,000 nodes with a configured neighbor limit of 5. This created networks with an average path length of ca. 4. The large network size and the low number of neighbors were chosen to produce a rather large average path length, which reduces the chance of matching data being stored directly on a neighbor node, thereby potentially making the results clearer. Within this network, we have

distributed our test data set according to our random data placement scheme as described above. The routing heuristic was set to a 10% failure probability, and the query evaluation parameters $weight$ and $improvement$ were set to 1.0 and 0.5, respectively. It should be noted that these settings were chosen primarily to reduce the number of independent variables. Later, we will also perform an experiment to determine the influence of these parameters.

Within this testbed, we have then run our three test queries 100 times each for the four component configurations, representing all combinations for the movement and reordering components enabled or disabled. These test queries for all configurations resulted in 1,200 data points:

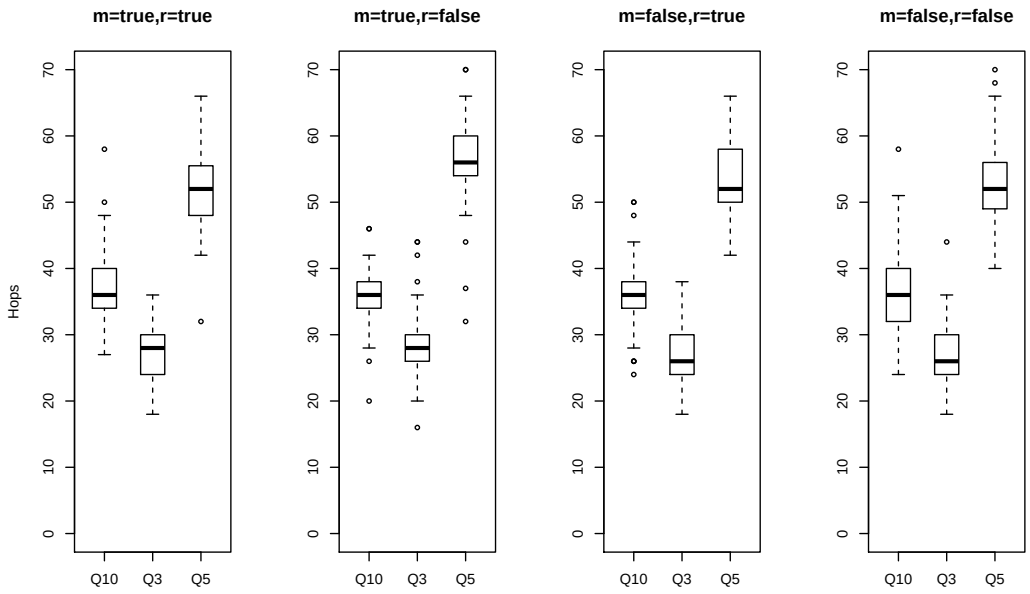


Figure 5.4: Component Effectiveness – Hop Count

The impact of en- and disabling components on the hop count is depicted in Fig. 5.4, each configuration has its own panel, and for each query a box plot over the hop count required to evaluate the query is given. All box plots in this chapter display the box over the lower and upper quartiles of the observed data, with the whiskers extending to the observations 1.5 times the inter-quartil range from the box border at most. More extreme values are

displayed as outliers. The bold horizontal line represents the median observation. Box plots provide a graphical representation for two important statistical measures, the expected value, which is approximated by the median, and the variance, which is shown by the size of the box.

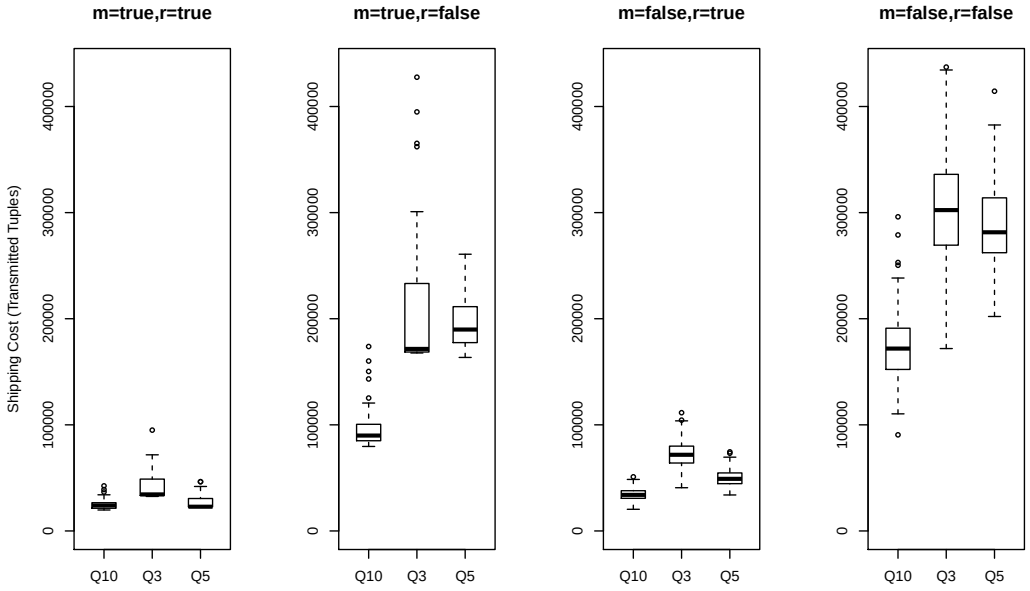


Figure 5.5: Component Effectiveness – Shipping Cost

We can observe how the different configurations have little effect on both the median value as well as the variance of the hop count results. This is due to the fact that the query evaluation process needs to visit all nodes where data matching its operators is stored. Also, due to the random data placement the return trip to the originating node and the outbound trip to the next node does not appear to be longer than the direct connection used when the evaluation process is moving.

However, an entirely different result can be seen in Fig. 5.5. Using the same layout as the previous plot, the total shipping cost (transmitted tuples) for the different configurations and queries are plotted here: We can observe that the runs with both the movement as well as the reordering component had the best result. Second, but close came the configuration where

the movement was disabled, and the reordering component enabled. The configurations with disabled reordering showed the highest median costs as well as the greatest variance, both not desirable from an efficiency standpoint.

These results are averaged over all queries run in Table 5.4. We can see the average median shipping cost differing by around a factor of 10 between the experiment with both components enabled and both components disabled.

| | (m=true,r=true) | (m=false,r=true) | (m=true,r=false) | (m=false,r=false) |
|---|---|---|---|---|
| min | 19,756 | 20,412 | 79,636 | 90,589 |
| **mean** | **31,231** | **52,466** | **164,393** | **254,514** |
| median | 28,162 | 49,060 | 169,620 | 266,488 |
| max | 95,032 | 111,456 | 427,604 | 437,152 |

Table 5.4: Component Effectiveness – Shipping Cost Measures

From these relative observations, we can conclude that the two main components of our approach, the movement of the query evaluation process through our network as well as the constant re-optimization both contribute to lower shipping costs. For the remainder of this chapter, both components will therefore remain enabled.

### 5.4.3 Network Size Impact

One of the goals for this work was the creation of a fully distributed query processing approach. One of the reasons for this approach being fully distributed was the potential gain in scalability. In this experiment, we analyze the relationship between the network size and the shipping costs. From our stochastic analysis in Section 4.6.1, we have determined that the expected costs are linear to the average path length inside the network, or logarithmic to the number of nodes for a certain connectivity. We predict this hypothesis to be true.

In this experiment, we have run our benchmark on seven network sizes between 100 and 5,000 nodes. Since we have predicted a logarithmic relationship, we assume that this range is sufficient to show this behavior. To also remove the effect of random starting nodes, data

placement and network structure, we have recreated the network setup 100 times for each network size and re-run the benchmark. For each query run, we have measured its total shipping cost. Overall«, 2,100 data points were collected for this experiment.

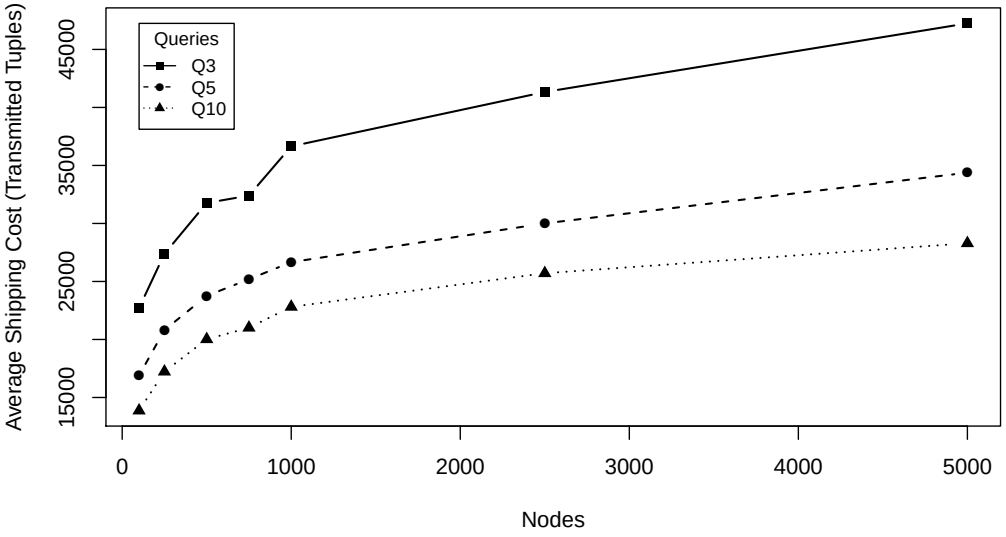


Figure 5.6: Environment Impact – Network Size – Query Averages

Fig. 5.6 shows these results. For each query, we have averaged the total shipping cost to remove random effects. We can observe the development of shipping costs and network size being consistent between queries. The logarithmic shape of the shipping cost development with regard to the network size gives a first indication that our expectation of a logarithmic relationship is accurate. A statistical analysis of this relationship is given later in this section.

Apart from the average, we are also interested in the statistical distribution of the shipping cost over the 100 repeated experiments for all three queries at once. Box plots for each network size are given in Fig. 5.7. From these plots, we can observe consistent results. The median values are increasing monotonic with the network size, and the small boxes containing 50% of all observations indicate little variance. Therefore, using the average values for further study is considered appropriate.

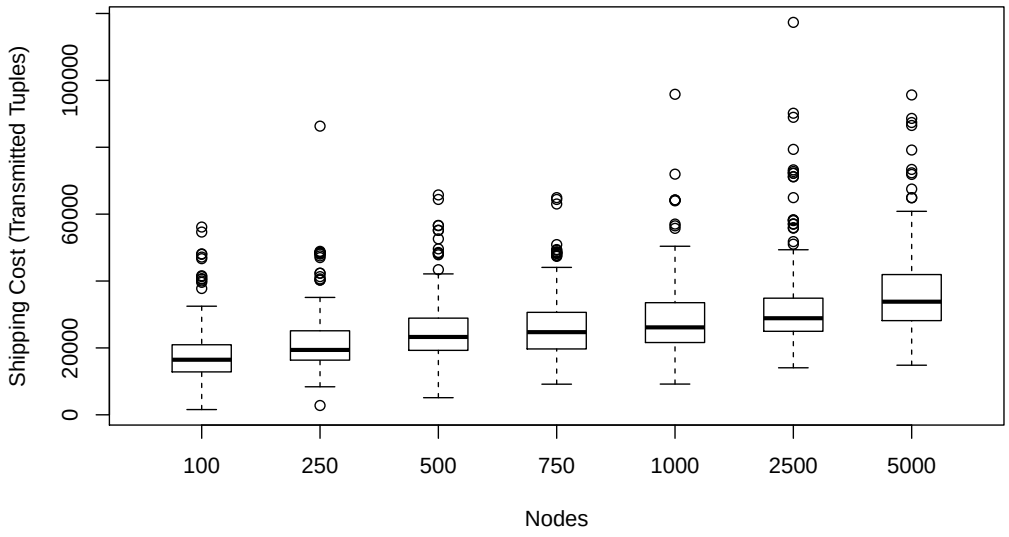


Figure 5.7: Environment Impact – Network Size – Overall Distribution

| Query | Distribution | Fitted Function | Correlation | Lack-of-Fit |
|---|---|---|---|---|
| Q3 | Logarithmic | $12012.91 * log(nodes)$ | 0.99 | 4% |
| Q5 | Logarithmic | $8892.14 * log(nodes)$ | 0.98 | 1% |
| Q10 | Logarithmic | $7457.44 * log(nodes)$ | 1 | 2% |
| Q3 | Linear | $4.31 * nodes + 27997.82$ | 0.91 | 16% |
| Q5 | Linear | $2.95 * nodes + 21124.39$ | 0.90 | 19% |
| Q10 | Linear | $2.43 * nodes + 17770.34$ | 0.87 | 24% |
| All | Logarithmic | $9454.16 * log(nodes)$ | 0.99 | 4% |
| All | Linear | $3.23 * nodes + 22297.51$ | 0.89 | 20% |

Table 5.5: Distribution Test – Network Size

To determine whether the shipping costs were dependent on the network size in a logarithmic way, we have performed a two-hypothesis statistical tests on the results. Table 5.5 shows the results of these tests. We have tested two hypotheses, first, that the relationship between network size and shipping cost is linear, and second, that this relationship is logarithmic. To

test this, we have fitted both a linear and a logarithmic function to the average shipping cost for each network size using the nonlinear-least-squares method [Bates and Watts, 1988]. We have then calculated the Pearson correlation and the lack-of-fit test [Neter et al., 1985], to test the quality of the fitted curve. We have performed this for both the average data from the individual queries as well as for all queries together.

From the results shown in the table, we can see that we can accept the logarithmic relationship and reject the linear relationship between network size and shipping cost with at least a 10% confidence level. We can therefore confirm our hypothesis of a logarithmic relationship being present between network size and shipping cost for this experiment.

### 5.4.4 Evaluation Efficiency

In the single-element retrieval experiment in Section 5.3, we have observed a profound impact of the number of hops required to retrieve a single element by the routing failure probability $p_f$. Since processing a query requires visiting each location with potentially matching data, we hypothesize that processing the queries from our benchmark would show similar susceptibility to this parameter.

From our stochastic analysis in Section 4.6.1, we have predicted a strong relationship between the routing failure probability and the average shipping cost. In this experiment, we test this prediction by running our benchmark on a single network with 1,000 nodes. We have performed 100 repetitions of our benchmark with the routing failure probability parameter set to 0, 0.1, 0.2, 0.3 and 0.4. For each run, we have collected the total shipping cost. In total, 1,200 data points were recorded.

As an example, we present the development of the shipping cost for Q10 in Fig. 5.8 for a single query evaluation run. To reiterate, Q10 finds the top 20 customers, where the lost revenue was greatest due to the customer returning parts. The horizontal axis represents the subsequent steps the query evaluation took through the network. On each step, we print the current shipping cost of the query and its intermediate results on the left vertical axis. Also, the cumulative shipping cost for the entire query evaluation is shown on the right vertical axis. We can observe new data arriving at step 4, 9, 14 and 17. Our query optimization has correctly scheduled the retrieval of large sets of tuples late in the process.

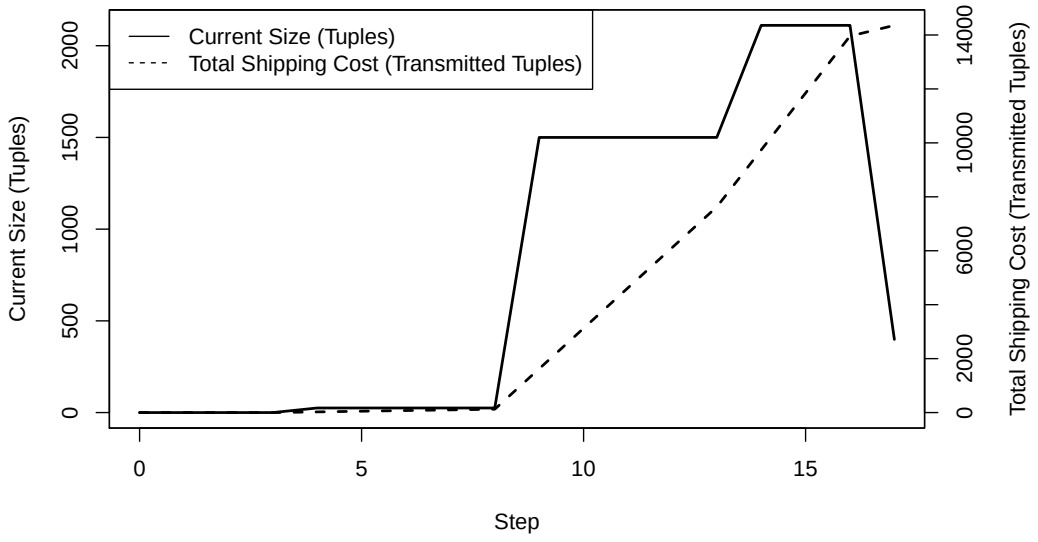


Figure 5.8: Query Shipping Cost Development – Query Q10

To compare our results, we have also calculated the shipping cost for the optimal plan by enumerating and evaluating all permutations of the query plans and evaluation orders for the three queries. This was performed in the same simulated network as the subsequent queries, thus making a comparison with the results of our approach possible. The optimal shipping cost for each query are printed in Table 5.6.

| Query | Optimal Shipping Cost |
|---|---|
| Q3 | 25,459 |
| Q5 | 18,263 |
| Q10 | 12,755 |

Table 5.6: Efficiency Test – Optimal Shipping Cost

The comparison between the average shipping cost and the pre-determined theoretical optimum is shown in Table 5.7. For the different settings of the routing failure probability parameter $p_f$, we see the average shipping cost over all query evaluation processes for this

configuration. Through a division of this value by the optimal shipping cost from Table 5.6, we are able to determine the average overhead created through variations in the query evaluation process, which are mainly influenced by the $p_f$ parameter. It is this table that quantifies the trade-off between routing accuracy and the efficiency of MMQP with regards to the shipping cost.

| $p_f$ | Query | Average | Overhead Factor |
|---|---|---|---|
| 0 | Q3 | 41,732 | 1.6 |
| 0 | Q5 | 24,895 | 1.4 |
| 0 | Q10 | 14,873 | 1.2 |
| 0.1 | Q3 | 40,476 | 1.6 |
| 0.1 | Q5 | 28,110 | 1.5 |
| 0.1 | Q10 | 18,456 | 1.4 |
| 0.2 | Q3 | 52,980 | 2.1 |
| 0.2 | Q5 | 35,259 | 1.9 |
| 0.2 | Q10 | 23,523 | 1.8 |
| 0.3 | Q3 | 61,997 | 2.4 |
| 0.3 | Q5 | 44,580 | 2.4 |
| 0.3 | Q10 | 30,429 | 2.4 |
| 0.4 | Q3 | 97,039 | 3.8 |
| 0.4 | Q5 | 64,690 | 3.5 |
| 0.4 | Q10 | 48,117 | 3.8 |

Table 5.7: Efficiency Test – Overhead Shipping Cost

From these results, we can observe a strong influence of the $p_f$ parameter to the average overhead. For example, a routing failure probability of 0.4 created on average a overhead between 3.5 and 3.8. Whether these values are acceptable again depends on the trade-off taken in the construction of the DSS, similar to what has been observed in Section 5.3. For networks with a lower failure probability such as for example 0.1 or 10%, the observed average

overhead only ranged within 1.4 and 1.6, which – given the fully distributed optimization of the query plans – is well below the threshold for general nearest neighbor TSP optimizations discussed in Section 4.6.1. Here, the advantage of our predictive and task-aware optimization approach becomes visible.

### 5.4.5 MMQP Environment and Parameter Impact

Apart from the routing failure probability $p_f$, our query processing approach is influenced by a number of other parameters. All these parameters control input values for our cost model presented in Section 4.3. The hypothesis for this experiment was that these parameters also have an influence on the total shipping cost, as bad decisions by the cost model will reflect in sub-optimal paths through the network and thus higher shipping costs. In particular, the following parameters were tested:

- Distance heuristic error $p_d$, tested values between 0.01 and 0.9
- Selectivity heuristic error $p_s$, tested values between 0.01 and 0.9
- Minimum improvement, tested values between 0.1 and 0.9
- Estimated future cost weight, tested values between 0 and 1000.

These experiments were carried out very similar to the previous two experiments. All configuration values except one were set to common-sense values, and the parameter to be tested was varied. For each configuration, our benchmark test was repeated 100 times. Furthermore, the network size was static at 1,000 nodes and the routing error probability was configured to 0.1. It should be noted that the tested parameter setting of 0 for the future weight cost is also a test of whether the future cost estimation has any effect on the shipping cost.

The experiments for distance and selectivity error as well for minimum improvement showed no influence at all for these parameters. An inspection of the optimized queries revealed their basic read operator selectivity to be very distinct, differing by orders of magnitude. Table 5.8 shows the cardinality for each read operator for our test data set for

| Query | Cardinalities |
|---|---|
| Q3 | 1,500 – 15,000 – 60,175 |
| Q5 | 5 – 25 – 100 – 1,500 – 15,000 – 60,175 |
| Q10 | 25 – 1,500 – 15,000 – 60,175 |

Table 5.8: Parameter Test – Query Read Cardinality

each query tested. We assume this being the reason for the absence of an observable influence of the first three parameters tested. However. extending our benchmark with queries that provoke them could test their influence.

| | Q3 | Q5 | Q10 |
|---|---|---|---|
| 0 | **179,004** | **288,093** | **48,440** |
| 1 | 38,990 | 25,523 | 22,162 |
| 10 | 37,961 | 27,638 | 22,848 |
| 100 | 38,564 | 26,669 | 23,608 |
| 1000 | 38,553 | 27,418 | 22,427 |

Table 5.9: Parameter Test – Weight

The results for the test with varying future cost weight parameters are shown in Table 5.9. For each query and parameter setting, the rounded average shipping cost over the 100 repetitions are shown. To reiterate, the future cost parameter controls the impact of the estimated future shipping cost after simulating the evaluation of an operator to the query selection process. From the results, we can again observe no significant impact for the parameter settings between 1 and 1,000. The result for the setting of 0 however, which leads to MMQP completely ignoring the future cost, showed considerable higher shipping costs. From this, we can assume a beneficial effect of the future cost calculation and recommend a non-zero weight setting.

## 5.5 Summary and Conclusions

We have started this chapter with a rationale for the use of controlled experiments for both our models as well as our algorithms. We described the test environment we have implemented to provide fully controlled environment, the synthetic heuristics and the TPC-H test data set and associated queries. We have then shown experiments aimed at confirming the behavior of our simulated network structure with the predictions regarding the costs of single-key retrieval operations. As we were able to confirm our predictions regarding logarithmic efficiency, we continued with testing the effectiveness of our query processing approach. Since we were able to create results equivalent to a commercial database for our benchmark queries, we concluded that our approach is indeed effective in evaluating complex queries in a distributed environment. For the subsequent experiments, we used the total shipping cost also the basis of our cost model as a metric for query evaluation efficiency. In order not to include components which have no effect on this efficiency, we have tested the basic concepts of continuous reordering and evaluation movement, and found both to have a beneficial relative impact on the shipping cost. We showed how the shipping cost has a logarithmic relationship with the network size, which is an important precondition to its scalability to arbitrary network sizes. Furthermore, we have shown how the routing failure probability inherent in the coordination method used in the network has a profound impact on the shipping cost. Finally, our experiments with other parameters showed how our investment/gain approach with its future size estimation further reduces shipping costs.

From these results, we can confirm that it is possible to perform complex query processing on unreliable and error-prone network architectures with surprisingly low additional costs in terms of network traffic. Our results give an indication on what level of efficiency can be expected from a specific network architecture with associated routing error probability. The simulation environment, test data and raw and intermediate results for all experiments are available for download. Access information and documentation is given in Appendix A.2.

# 6 Conclusion

Over the previous chapters, we have covered complex query processing in fully distributed storage systems from various angles. We have started with discussing abstract goals for distributed storage in general based on previous literature. We could see how scalability is the most frequently cited goal for deploying distributed storage, but how economical considerations also clearly play a role. From the literature, we have then synthesized abstract goals for distributed storage, namely scalability, consistency and availability. These competing goals created a large solution space, where every single architecture leads to a specific set of compromises between between them. From reviewing representatives for different distributed storage architectures with their particular trade-offs, it was apparent that these compromises prohibit the creation of a "one-size-fits-all" distributed storage system, and therefore the choice of an architecture depends on the intended system environment and application requirements. Focusing on a single solution was therefore impossible, and the creation of an abstracting architecture was necessary.

We have then continued to review several popular data and access models for distributed storage systems. From our comparison, it became clear how storing structured data has considerable advantages, and how fine-grained access to the stored data through complex queries is clearly desirable. We have then presented the state of the art in distributed complex query processing on structured data. From this literature review, we were able to observe how closely tied previous approaches are to the underlying network architecture and coordination model. We argued that this is a major hindrance for development in this area, as these approaches cannot readily be adapted to new network or coordination models, and how a comparison between them is very difficult. Indeed, several proposed distributed storage architectures do not provide support for complex queries at all, mainly due to fear

of possible repercussions on overall efficiency. If a method to predict the behavior and efficiency of complex query processing on the specific architecture would exist, these entry barriers would presumably be much lower. We have also seen how not only the process of query execution, but also the process of query optimization has to be distributed in a fully distributed environment.

This lead to our first research problem, the creation of an abstract network and coordination model as a basis for our further work on distributed storage. We have used the model of a fully connected but random network of connections between fully independent storage nodes. Between these nodes, we assumed the presence of a routing heuristic, which is able to forward requests for data containing a certain key to one of the nodes the current node is connected to. To allow for non-exact coordination methods, this routing method contained a certain error probability, which is an environment parameter expressing how likely the routing method is to take an incorrect decision. We have demonstrated the flexibility of this model by applying it to representative architectures from the three main classes of centralized, structured and unstructured approaches.

We have found that our model was able to express all of them and deem it being suitable for the abstract expression of distributed processes that can then be adapted to a specific architecture choice. We have also presented a query and data model based on the relational model, but without the need for a central schema and therefore with tuples as a smallest coherent unit. From discussing the question of where individual data items should be stored in a distributed system, we have seen how locality in same-key data placement is a requirement for efficient and complete retrieval results. Through an average-case stochastic analysis, we have seen how retrieval of any data item within this network model has logarithmic complexity with regard to the number of hops, at least if the employed coordination mechanism is more likely to correctly route operations than not to do so. Therefore, we were able to assume sufficient performance for retrieval operations in our network model and continued to discuss a higher algorithm.

We have then introduced Mutable Moving Query Plans, which receive a complex query on any node that is part of the network, and them embarks on a journey through the network while constantly re-optimizing the query plan itself as well as already collecting intermediate

results. Finished results are returned to the node where the process originated, and results can then be delivered to waiting applications. However, every optimization needs a metric to determine costs with, we have therefore also developed such a cost model, which is based on the number of transmitted tuples or network traffic. We have presented this query processing approach with algorithmic definitions on top of our network model. We have again analyzed the average-case complexity with regards to our cost model, and found that a logarithmic relationship with the network size can also be expected, which is sufficient for the scalability of our approach.

However, we were not content with analytical considerations. Therefore, we have also performed a set of controlled experiments. For our experiments, we have implemented a simulation environment based on our proposed network model as well as our MMQP proposal. Using the TPC-H test data and query benchmark, we were able to confirm both effectiveness of our query processing concept as well as the predicted logarithmic relationship between the network size and query execution efficiency to our traffic-based cost model. We have also determined the impact of the routing error probability to the execution efficiency. From these results, we were able to confirm that it is possible to perform complex query processing in our abstract-yet-versatile network model with surprisingly low additional costs in terms of network traffic. Also, our experimental results confirm that our network model and data placement scheme can form a minimal nucleus of properties a DSS has to exhibit in order to make complex query processing possible. Our results can also give an indication on what degree efficiency can be expected from a specific network architecture with associated routing error probability.

## Future Work

Our promising results from discussing the main issues of distributed complex query processing warrant future work in many directions. There, contributions would also add support to our main goal of promoting complex query support in distributed storage. For example, we have not considered the transactional paradigm at all. The very likely case of data being changed while a query is processed does contain a huge number of additional challenges,

and it is unclear whether guaranteeing for example the ACID properties in our model is possible at all. An immediate precondition of discussing these consistency issues would be to develop a notion of completeness of query responses. While we do not believe the full information retrieval abstraction of precision and recall is necessary or even suited to this case, completeness would have to be discussed further, particularly with respect to different data placement schemes.

Another area of work are the data placement schemes that govern the distribution of data stored using an equivalent key. We have already shown how some degree of locality is required for retrieval efficiency, but have assumed an key-node identity for the main parts of this work. However, this is of course a simplified view, and a discussion on what different data placement schemes are conceivable, and what impact they would have on both efficiency and completeness would be highly interesting. In this context, methods that would change the network structure of the storage network would be also possible. For example, we have proposed that misplaced data in the network should be moved to the correct location. However, creating a new network connection might be more efficient.

In query optimization, we have seen how estimating the result set cardinality of operators not yet evaluated is impossible in general, and how methods to estimate them are based on statistics of previous query executions. In the interest of experimental independence, we had not considered these approaches further, also, in our network model, these statistics would be scattered over all nodes and were not assumed to be very accurate. However, future work in this area could further shorten the distance between our fully distributed and continuously optimizing MMQP proposal and previous conventional centralized approaches.

# A Appendix

## A.1 TPC-H Schema, Queries and Translation

As mentioned in Section 5.2, we have used TPC-H data and queries to test MMQP. Here, we give some details about the TPC-H schema and the test queries used. We also describe the translation of the queries into the query model we have presented in Section 3.5. The TPC-H data set is built on a classical trade scenario. A company sells items, which they procure from suppliers. Customers send orders, which can contain many items. A relational database schema for this is given in Fig. A.1. For each relation, a table of the columns is given. The arrows denote foreign keys. Data for this schema can be generated using the `dbgen` data generator. The data generator uses a so-called scaling factor to determine how many instances of each schema should be created. The amount of instances for a scaling factor is given under the relation name printed in bold, for example, the supplier table contains 10,000 times the scaling factor instances. For our experiments, we have used a scaling factor of 0.01. The generated instances were translated into our data model using the translation scheme described in Section 3.3.

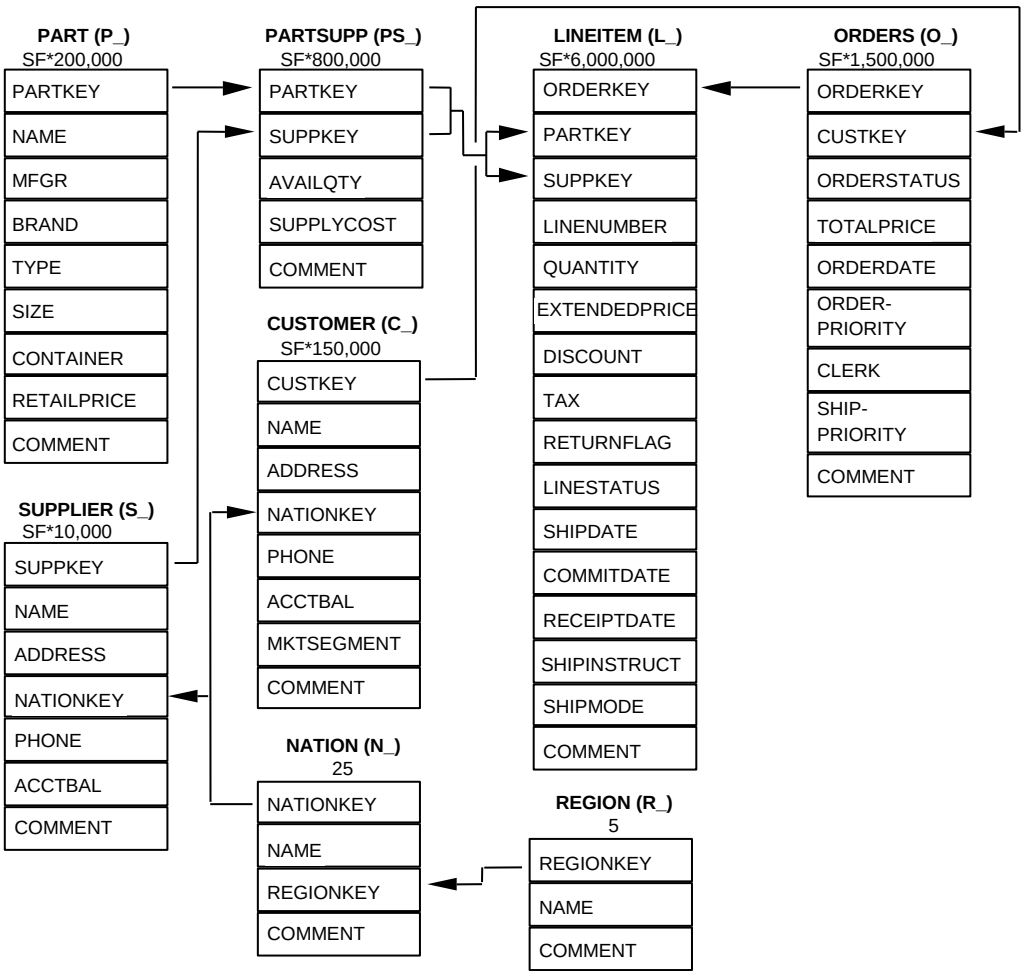


Figure A.1: TCP-H – Schema [TPC, 2011]

### A.1.1 Query 3: Shipping Priority

The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of `l_extendedprice * (1 - l_discount)`, of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more

than 10 unshipped orders exist, only the 10 orders with the largest revenue are listed. [TPC, 2011]

Listing A.1 contains the SQL representation of this query. The operator tree is shown in Fig. A.2. Three selections with six basic read operations and two joins are required to collect all the input data, which are then aggregated in the root.

Listing A.1: TCP-H Query 3 – SQL

```
SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue,
  o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey  AND
  l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate
  > '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue DESC, o_orderdate
```

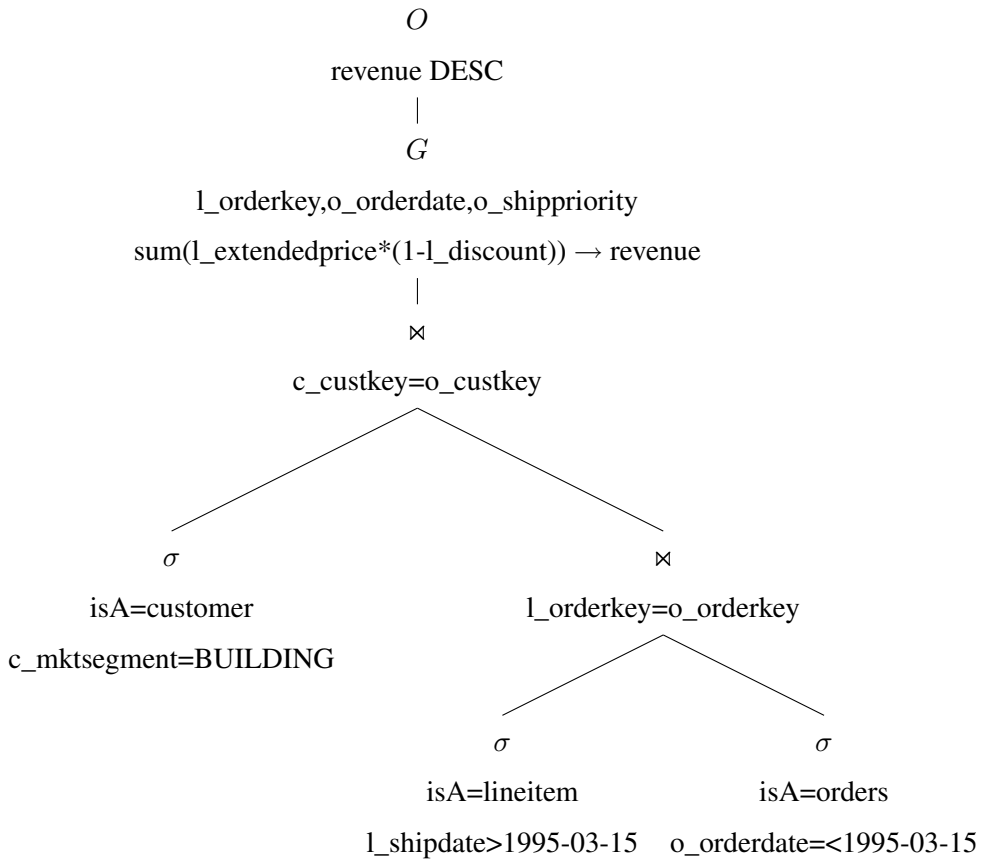


Figure A.2: TCP-H Query 3 – Tree Representation

### A.1.2 Query 5: Local Supplier Volume

The Local Supplier Volume Query lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. The query is run in order to determine whether to institute local distribution centers in a given region. The query considers only parts ordered in a given year. The query displays the nations and revenue volume in descending order by revenue. Revenue volume for all qualifying lineitems in a particular nation is defined as `sum(l_extendedprice * (1 - l_discount))`. [TPC, 2011]

Listing A.2 contains the SQL representation of this query. The operator tree is shown in Fig. A.3. Six selections with nine basic read operations and five joins are required to collect all the input data, which are then aggregated in the root.

Listing A.2: TCP-H Query 5 – SQL

```
SELECT n_name, SUM(l_extendedprice*(1-l_discount)) AS revenue
FROM customer, orders, lineitem, supplier, nation, region
WHERE l_orderkey = o_orderkey AND c_custkey = o_custkey AND l_suppkey =
   s_suppkey AND c_nationkey = s_nationkey AND s_nationkey = n_nationkey
   AND n_regionkey = r_regionkey AND r_name = 'ASIA' AND o_orderdate >=
  '1994-01-01' AND o_orderdate < '1995-01-01'
GROUP BY n_name
ORDER BY revenue DESC
```

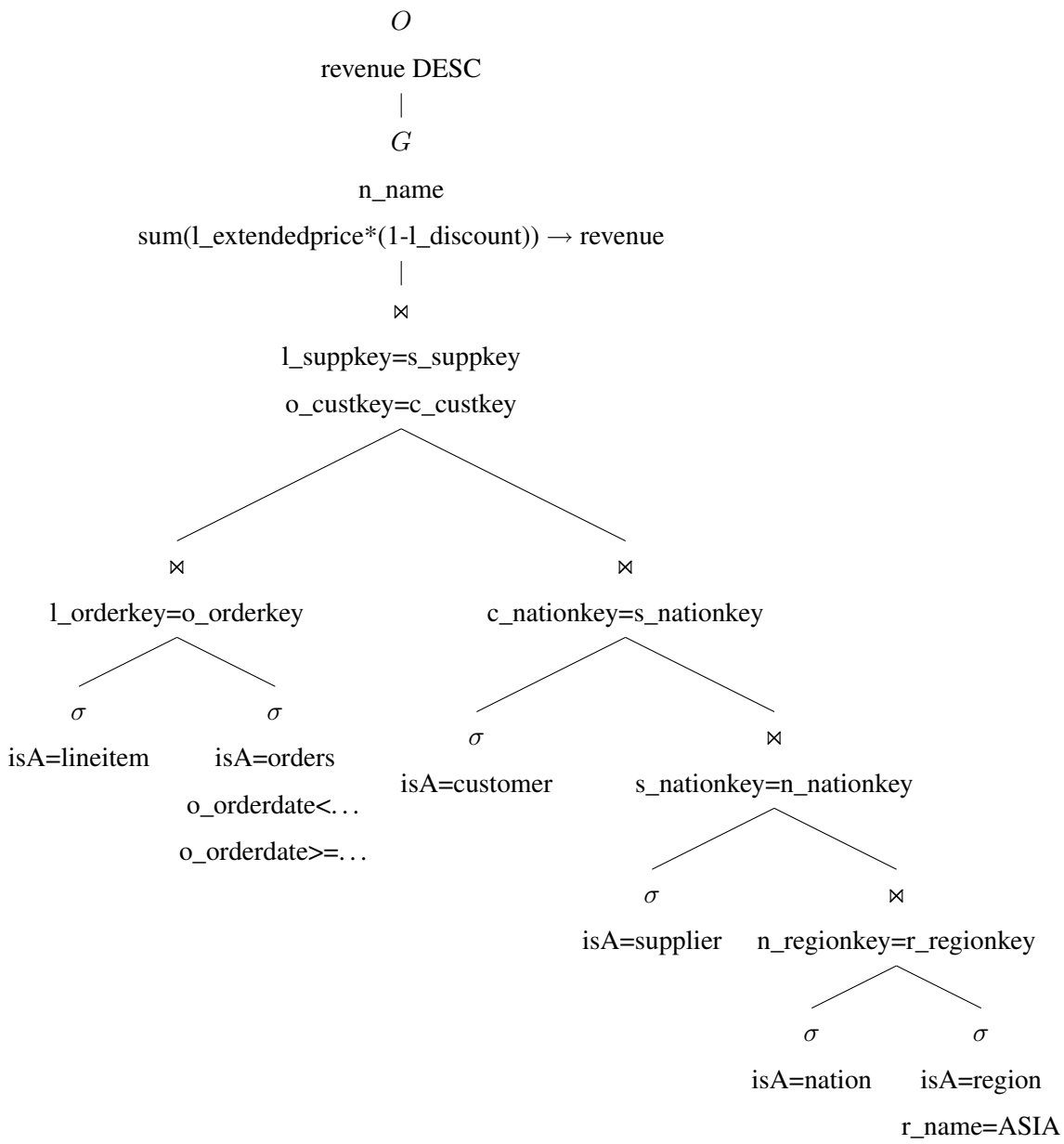


Figure A.3: TCP-H Query 5 – Tree Representation

### A.1.3 Query 10: Returned Item Reporting

> The Returned Item Reporting Query finds the top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, phone number, account balance, comment information and revenue lost. The customers are listed in descending order of lost revenue. Revenue lost is defined as `sum(l_extendedprice * (1 - l_discount))` for all qualifying lineitems. [TPC, 2011]

Listing A.3 contains the SQL representation of this query. The operator tree is shown in Fig. A.4. Four selections with seven basic read operations and three joins are required to collect all the input data, which are then aggregated again in the root.

Listing A.3: TCP-H Query 10 – SQL

```
SELECT c_custkey, c_name, SUM(l_extendedprice*(1-l_discount)) AS
  revenue, c_acctbal, n_name, c_address, c_phone, c_comment
FROM customer, orders, lineitem, nation
WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate
   >= '1993-10-01' AND o_orderdate < '1994-01-01' AND l_returnflag = 'R'
   AND c_nationkey = n_nationkey
GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
  c_comment
ORDER BY revenue DESC
```

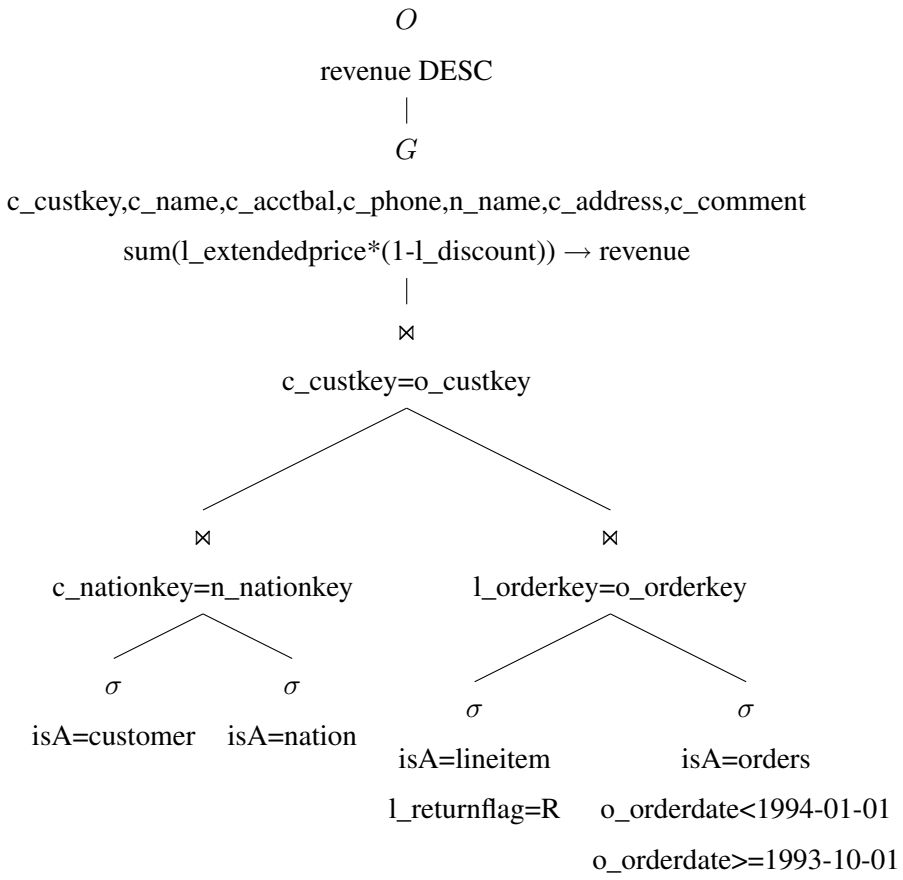


Figure A.4: TCP-H Query 10 – Tree Representation

## A.2 Experimental Environment and Results

As mentioned, we have implemented our abstract network model and our MMQP approach as in order to perform our controlled experiments in a simulation program. In this section, we describe all components required to repeat or adapt our experiments. The general work flow consists of executing a Java program, which writes its results into a Comma-Separated Values (CSV) file. This CSV file is then used by scripts in the statistics environment R[1] to produce aggregated tables and graphs. The simulation program is implemented in the Java programming language. The language specification version the program conforms to is

[1] `http://www.r-project.org/`, accessible as of 2012-07-29

version 1.6.0, which was the most recent version at the time of implementation. Dependencies are handled using the Maven build tool, version 2.2.1[2]. The source code is available on the author's web page[3]. This package already contains a executable Java JAR file in the `target/` directory. To recompile, extract the package, use a terminal to change into the extracted directory, and run the console command `mvn install`. This command should create the executable JAR file `target/MMQP-*.jar`. This process is also required after changing the simulation parameters for the experiments. Both raw and processed results used in this thesis are also contained in the archive.

### A.2.1 Single-Element Retrieval Experiment

The Single-Element retrieval experiment (described in Section 5.3) tests how many hops are required to retrieve a single element from all nodes in the network. This experiment is implemented in the Java class `org.fuberlin.nbi.mmqp.experiments.Routing`. The following parameters can be set inside the Java source:

- `networkSizes` – Set of network sizes the experiment should be run in
- `minNeighbors` – Minimum amount of neighbors, controls the network bootstrap algorithm termination criterion.
- `testProbabilities` – Set of routing failure probabilities to test.
- `repetitions` – Amount of repetitions for each permutation of the three previous parameters.

After recompilation, the experiment can be run by executing the following shell command in the MMQP directory:

```
java -cp target/*.jar org.fuberlin.nbi.mmqp.experiments.Routing .
```

The process creates a results CSV file `routing.csv`. This file contains the following data fields:

[2] `http://maven.apache.org/`, accessible as of 2012-07-29

[3] `http://hannes.muehleisen.org/mmqp/MMQP.zip`, accessible as of 2012-07-29

1. `neighbors` – Amount of neighbors per node
2. `diameter` – Network diameter (greatest distance between nodes)
3. `degree` – Average number of connections per node
4. `avgPathLength` – Average path length
5. `nodes` – Amount of nodes in the network
6. `errorProb` – Routing failure probability $p_f$
7. `steps` – Amount of steps taken to retrieve item
8. `run` – Run number
9. `success` – Whether the item was found

This file can be analyzed with the R script `experiments/routing/routing.R`. This script produces graphs of the network size against the hop count for ten neighbors and also for the aggregated value of average path length. In addition, the script outputs the results of the statistical comparison of our prediction and the experimental results.

### A.2.2 Query Evaluation Effectiveness Experiment

The query evaluation effectiveness experiment (described in Section 5.4.1) tests whether MMQP produces correct results when compared with the off-the-shelf relational database HSQLDB[4]. This experiment is implemented in the Java class `org.fuberlin.nbi.mmqp.experiments.Effectiveness`. This experiment has no parameters. However, if additional queries should be run, these can be set in the Java class `org.fuberlin.nbi.mmqp.TPC`. After recompilation, the experiment can be run by executing the following shell command in the MMQP directory:

```
java -cp target/MMQP-*.jar org.fuberlin.nbi.mmqp.experiments.
  Effectiveness .
```

[4] `http://www.hsqldb.org`, accessible as of 2012-07-29

This process creates a set of CSV files: `effectiveness.csv` contains a list of the used query identifiers along with the result set sizes for both MMQP and HSQLDB. In addition, the result sets for each query and system are also written to CSV files, e.g. `Q5-hsql.csv` for the query `Q5` on the `HSQLDB` system. Further analysis was performed by manually comparing the result sets and their size figures.

### A.2.3 Component Effectiveness Experiment

The component effectiveness experiment (described in Section 5.4.2) tests whether enabling movement and constant re-optimization for MMQP is visible in the total shipping cost. This experiment is implemented in the Java class `org.fuberlin.nbi.mmqp.experiments.Components`. The following parameters can be set inside the Java source:

- `nodes` – Set of network sizes the experiment should be run in
- `neighbors` – Minimum amount of neighbors, controls the network bootstrap algorithm termination criterion.
- `testProbabilities` – Set of routing failure probabilities to test.
- `repetitions` – Amount of repetitions for each permutation of the three previous parameters.
- `probability` – Error probability for routing, distance and size heuristics.
- `improvement` – Minimum improvement configuration parameter for MMQP, controls by how much an alternative plan has to be better than the current plan to be selected.

After recompilation, the experiment can be run by executing the following shell command in the MMQP directory:

```
java -cp target/*.jar org.fuberlin.nbi.mmqp.experiments.Components .
```

The process creates a results CSV file `components.csv`. This file contains the following fields:

1. `traffic` – Total shipping cost incurred
2. `queryName` – Query
3. `movement` – Whether movement was enabled
4. `reordering` – Whether reordering was enabled
5. `steps` – Amount of steps taken to process query
6. `run` – Run number
7. `success` – Whether the query evaluation was successful

The CSV file can be analyzed with the R script `experiments/components/components.R`. This script produces box plots of the different configurations for each query, both for total hop count as well as shipping cost according to our cost model.

### A.2.4 Parameter Impact Experiment

The parameter experiment (described in Section 5.4.3 and Section 5.4.5) tests the impact of various parameters to the total shipping cost. Parameters tested are network size, routing error probability, distance and size heuristic error, MMQP minimum improvement and future cost weight. This experiment is implemented in the Java class `org.fuberlin.nbi.mmqp.experiments.Parameters`. The following parameters can be set inside the Java source:

- `defaultNodes` – Network size for the experiments where this parameter is not tested.
- `defaultNeighbors` – Minimum amount of neighbors, controls the network bootstrap algorithm termination criterion.

- `defaultProb` – Error probability of routing, size and distance heuristics when the respective parameter is not tested.
- `defaultWeight` – Future size weight factor for MMQP for experiments where this parameter is not tested.
- `defaultImprovement` – Minimum improvement configuration parameter for MMQP, controls by how much an alternative plan has to be better than the current plan to be selected.
- `testProbabilities` – Set of failure probabilities to test.
- `testImprovements` – Set of MMQP improvement factors to be tested.
- `testNetworkSizes` – Set of network sizes to be tested.
- `testWeights` – Set of MMQP future cost weighing factors to be tested.
- `repetitions` – Amount of repetitions for each permutation of the three previous parameters.

After recompilation, the experiment can be run by executing the following shell command in the MMQP directory:

```
java -cp target/*.jar org.fuberlin.nbi.mmqp.experiments.Parameters .
```

The process creates a results CSV file `parameters.csv`. This file contains the following fields:

1. `routingError` – Routing heuristic error rate setting
2. `weight` – Future cost weight setting
3. `test` – Parameter being tested
4. `queryName` – Query
5. `avgPathLength` – Average path length in network

6. `neighborLimit` – Maximum number of neighbors per node
7. `steps` – Amount of steps taken to process query
8. `run` – Run number
9. `traffic` – Total shipping cost created
10. `selectivityError` – Selectivity heuristic error rate setting
11. `distanceError` – Distance heuristic error rate setting
12. `nodes` – Number of nodes
13. `improvement` – Minimum improvement parameter setting
14. `success` – Whether the query evaluation was successful

The file can be analyzed with the R script `experiments/parameters/parameters.R`. For each parameter tested, this script generates three files: a box plot of the single experiment results grouped by parameter setting, a distribution plot showing minimum, mean and maximum results, and a plot showing results for each query run.

### A.2.5 Evaluation Efficiency Experiment

The evaluation efficiency experiment (described in Section 5.4.4) tests the efficiency of MMQP against a perfect scenario for different error probabilities.This experiment is implemented in the Java class `org.fuberlin.nbi.mmqp.experiments.Efficiency`. The following parameters can be set inside the Java source:

- `nodes` – Network size for the experiments.
- `neighborLimit` – Minimum amount of neighbors, controls the network bootstrap algorithm termination criterion.
- `weight` – Future size weight factor for MMQP.

- `improvement` – Minimum improvement configuration parameter for MMQP, controls by how much an alternative plan has to be better than the current plan to be selected.
- `repetitions` – Amount of repetitions for each permutation of the three previous parameters.
- `testProbabilities` – Set of heuristics failure probabilities to test.

After recompilation, the experiment can be run by executing the following shell command in the MMQP directory:

```
java -cp target/*.jar org.fuberlin.nbi.mmqp.experiments.Efficiency .
```

The process creates a results CSV file `efficiency.csv`. This file contains the following fields:

1. `queryName` – Query run
2. `errorProb` – Error probability for all heuristics
3. `minTraffic` – Minimum shipping cost
4. `perfectTraffic` – Optimal shipping cost
5. `averageTraffic` – Average shipping cost
6. `maxTraffic` – Maximum shipping cost

This file can be analyzed with the R script `experiments/efficiency/efficiency.R`. This script generates plots showing the best possible shipping cost, and the minimum, mean and maximum shipping cost achieved by MMQP for the different error probability settings.

# Bibliography

TPC benchmark H standard specifications, Nov 2011. Revision 2.14.3.

M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 126–140, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63575-0.

R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4.

A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286 (5439):509–512, 1999.

L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

D. M. Bates and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Wiley, New York, 1988.

D. Battré, F. Heine, A. Höing, and O. Kao. On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores. In G. Moro, S. Bergamaschi, S. Joseph, J.-H. Morin, and A. M. Ouksel, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 4125 of *Lecture Notes in Computer Science*, pages 343–354. Springer, 2006. ISBN 978-3-540-71660-0.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

K. Bharath-Kumar and J. M. Jaffe. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications*, COM-31:343–351, 1983.

K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th Conference on Very Large Databases*, VLDB '84. Morgan Kaufman, Aug. 1984.

E. A. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 7–10, NY, July 16–19 2000. ACM Press.

I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Distributed queries and query optimization in schema-based P2P-systems. In K. Aberer, V. Kalogeraki, and M. Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2003. ISBN 3-540-20968-9.

M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. MAAN: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Operating Systems Design and Implementation*, OSDI '06, pages 205–218. USENIX Association, 2006.

Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, editors, *Proceedings of the 2003 conference on applications, technologies,architectures, and protocols for computer communications*, SIGCOMM '03, pages 407–418. ACM, 2003. ISBN 1-58113-735-4.

C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211 – 229, 2000. ISSN 0304-3975.

M. C. Chu-Carroll. *Code in the Cloud: Programming Google AppEngine*. Pragmatic Bookshelf (O'Reilly), Sebastopol, CA 95472, 2011. ISBN 9781934356630.

E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-14192-2.

G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2002. ISBN 978-0-201-61918-8.

A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *22th International Conference on Distributed Computing Systems (22th ICDCS'02)*, pages 23–, Vienna, Austria, July 2002. IEEE.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (21st SOSP'07)*, pages 205–220, Stevenson, Washington, USA, Oct. 2007. ACM SIGOPS.

W. K. Dedzoe, P. Lamarre, R. Akbarinia, and P. Valduriez. Asap top-k query processing in unstructured p2p systems. In *Peer-to-Peer Computing*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-7141-6.

D. DeWitt and J. Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. M. Özsoyoglu. A complete translation from SPARQL into efficient SQL. In B. C. Desai, D. Saccà, and S. Greco, editors, *2009 International Database Engineering and Applications Symposium*, ACM International Conference Proceeding Series, pages 31–42. ACM, 2009. ISBN 978-1-60558-402-7.

P. Erdos and A. Rényi. On the evolution of random graphs. *Publications of the Mathematics Institute of the Hungarian Academy of Science*, 5:17–61, 1960.

A. Fikes. Storage architecture and challenges. Online, `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//university/relations/facultysummit2010/storage_architecture_and_challenges.pdf`, July 2010.

W. Fontijn and P. A. Boncz. AmbientDB: P2P data management middleware for ambient intelligence. In *IEEE International Conference on Pervasive Computing and Communications – Workshops*, PerCom '04 Workshops, pages 203–207. IEEE Computer Society, 2004.

J. C. Freytag. A rule-based view of query optimization. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, page 173, San Francisco, CA, May 1987.

A. Fronczak, P. Fronczak, and J. A. Hołyst. Average path length in random networks. *Physical Review E*, 70:056110 (Article Number), Nov 2004.

C. GauthierDickey and C. Grothoff. Bootstrapping of peer-to-peer networks. In *2008 Symposium on Applications and the Internet*, SAINT '08, pages 205–208. IEEE Computer Society, 2008. ISBN 978-0-7695-3297-4.

S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *2003 Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

G. Giakkoupis and V. Hadzilacos. On the complexity of greedy routing in ring-based peer-to-peer networks. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 99–108, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5.

S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. In B. Eaglestone, S. North, and A. Poulovassilis, editors, *2002 British National*

*Conference on Databases*, volume 2405 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2002. ISBN 3-540-43905-6.

S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *International Workshop on the Web and Databases*, WebDB '01, pages 31–36, 2001.

I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-383-9.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, Dec. 1983. ISSN 0360-0300.

P. Hall, P. Hitchcock, and S. Todd. An algebra of relations for machine computation. In *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages*, pages 225–232, Jan. 1975.

B. Harangsri. *Query Result Size Estimation Techniques in Database Systems*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales.

Harren, Hellerstein, Huebsch, Loo, Shenker, and Stoica. Complex queries in DHT-based peer-to-peer networks. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 1, 2002.

S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In A. Fokoue, Y. Guo, and T. Liebig, editors, *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 10 2009.

P. Hayes. RDF semantics. World Wide Web Consortium, Recommendation REC-rdf-mt-20040210, Feb. 2004.

Y. He, H. Ren, Y. Liu, and B. Yang. On the reliability of large-scale distributed systems–A topological view. In *Proceedings of the 2008 International Conference on Parallel Processing*, ICPP '08, pages 165–172, Portland, OR, Sept. 2008. IEEE Computer Society.

M. Held and R. M. Karp. The travelling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

M. D. Hill. What is scalability? *Computer Architecture News*, 18(4):18–21, Dec. 1990.

F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Martí, and E. Cesario. The xtreemFS architecture - a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

D. Ilie, D. Erman, A. Popescu, and A. Nilsson. Measurement and analysis of gnutella signaling traffic. Technical report, School of Engineering - Dept. of Telecommunication Systems / Blekinge Institute of Technology.

I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58, Oct. 2008. ISSN 0360-0300.

D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. Wiley, Chichester, 1997.

V. Kantere, D. Tsoumakos, and N. Roussopoulos. Querying structured data in an unstructured P2P system. In A. H. F. Laender, D. Lee, and M. Ronthaler, editors, *Proceedings of the 6th annual ACM international workshop on Web information and data management*, WIDM '04, pages 64–71. ACM, 2004. ISBN 1-58113-978-0.

Karger, Lehman, Leighton, Levine, Lewin, and Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1997.

M. Karnstedt, K. Hose, and K.-U. Sattler. Distributed query processing in P2P systems with incomplete schema information. In Z. Bellahsene and P. McBrien, editors, *Third International Workshop on Data Integration over the Web*, DIWeb '04, pages 34–45, 2004.

G. Kokkinidis and V. Christophides. Semantic query routing and processing in P2P database systems: The ICS-FORTH SQPeer middleware. In W. Lindner, M. Mesiti, C. Türker, Y. Tzitzikas, and A. Vakali, editors, *Proceedings of the 2004 international conference on Current Trends in Database Technology*, volume 3268 of *Lecture Notes in Computer Science*, pages 486–495. Springer, 2004. ISBN 3-540-23305-9.

D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 2:48–50, 1956.

A. D. Kshemkalyani and M. Singhal. *Distributed Computing - Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997. ISBN 978-0-521-56067-2.

A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 395–406, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4.

Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing (ICS-02)*, pages 84–95, New York, June 22–26 2002. ACM Press.

V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 659–670, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8.

S. I. McClean, D. A. Bell, and F. J. McErlean. Heuristic methods for the data placement problem. *The Journal of the Operational Research Society*, 42(9):pp. 767–774, 1991. ISSN 01605682.

J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

H. Mühleisen. Query processing in a self-organized storage system. In *Proceedings of the VLDB PhD Workshop, co-located with 37th Intl. Conference on Very Large Databases (VLDB2011)*, 2011.

H. Mühleisen and K. Dentler. Large-scale storage and reasoning for semantic data using swarms. *Computational Intelligence Magazine, IEEE*, 7(2):32 –44, may 2012. ISSN 1556-603X.

H. Mühleisen, A. Augustin, T. Walther, M. Harasic, K. Teymourian, and R. Tolksdorf. A self-organized semantic storage service. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services (iiWAS2010)*, 2010.

H. Mühleisen, T. Walther, and R. Tolksdorf. A survey on self-organized semantic storage. *International Journal of Web Information Systems*, 7(3):205–222, 2011a.

H. Mühleisen, T. Walther, and R. Tolksdorf. Multi-level indexing in a distributed self-organized storage system. In *IEEE Congress on Evolutionary Computation*, pages 989–994. IEEE, 2011b. ISBN 978-1-4244-7834-7.

H. Mühleisen, T. Walther, and R. Tolksdorf. Data location optimization for a self-organized distributed storage system. In *Proceedings of the Third World Congress on Nature and Biologically Inspired Computing*, NaBIC '11, pages 176–182. IEEE, 2011c. ISBN 978-1-4577-1122-0.

A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Operating Systems Design and Implementation*, OSDI '02, 2002.

J. Neter, W. Wasserman, and M. H. Kutner. *Applied Linear Statistical Models*. Irwin, Homewood, Illinois, 1985.

C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In J. T.-L. Wang, editor, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110. ACM, 2008. ISBN 978-1-60558-102-6.

M. T. Özsu and P. Valduriez. Distributed database systems: Where are we now. *IEEE Computer*, 24:68–78, 1991.

M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

V. Papadimos and D. Maier. Mutant query plans. *Information and Software Technology*, 44 (4):197–206, 2002.

Peleg and Pincas. The average hop count measure for virtual path layouts. In *DISC: International Symposium on Distributed Computing*. LNCS, 2001.

A. Philippou and G. Michael. Verification techniques for distributed algorithms. In M. Shvartsman, editor, *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 172–186. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-49990-9.

H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible rule-based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, page 39, San Diego, CA, June 1992.

M. Placek and R. Buyya. A taxonomy of distributed storage systems. Technical Report GRIDS-TR-2006-11, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia.

K. R. Popper. *The Logic of Scientific Discovery*. Basic Books, New York, 1959.

E. Prud'Hommeaux and A. Seaborne. SPARQL query language for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115, Jan. 2008.

K. Ramamritham and P. K. Chrysanthis. A taxonomy of correctness criteria in database applications. *The VLDB Journal*, 5:85–97, 1996.

S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, 2001.

M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6:50–57, 2002. ISSN 1089-7801.

P. Rösch, K.-U. Sattler, C. von der Weth, and E. Buchmann. Best effort query processing in DHT-based P2P systems. In *Proceedings of the 21st International Conference on Data Engineering Workshops*, ICDEW '05, page 1186, 2005.

R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. *Innovations in Internetworking*, chapter Design and implementation of the Sun network filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988. ISBN 0-89006-337-0.

R. Segala. Verification of randomized distributed algorithms. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 232–260. Springer, 2001. ISBN 978-3-540-42479-6.

G. P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM. ISBN 0-89791-001-X.

W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

J. Silvestre. Economies and diseconomies of scale. In J. Eatwell, M. Milgate, and P. Newman, editors, *The New Palgrave: A Dictionary of Economics*. Palgrave Macmillan, 1987.

M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., New York, NY, USA, 1994. ISBN 007057572X.

I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, 2001.

M. Stonebraker. The case for shared-nothing. *IEEE Data Engineering Bulletin*, 9(1):4–9, Mar. 1986.

A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002. ISBN 0-13-088893-1.

S. Tang, H. Wang, and P. V. Mieghem. The effect of peer selection with hopcount or delay constraint on peer-to-peer networking. In A. Das, H. K. Pung, F. B.-S. Lee, and L. W.-C. Wong, editors, *Networking*, volume 4982 of *Lecture Notes in Computer Science*, pages 358–365. Springer, 2008. ISBN 978-3-540-79548-3.

Teradata. *DBC/1012 Data Base Computer, Concepts and Facilities*. Teradata Corporation, Los Angeles, CA, 1983.

F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proceedings of the 19th Conference on Very Large Databases*, VLDB '03, pages 333–344, 2003.

W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

C. Treijtel. AmbientDB: Complex query processing for P2P networks. In M. H. Scholl and T. Grust, editors, *Proceedings of the VLDB PhD Workshop, co-located with 29th*

*Intl. Conference on Very Large Databases (VLDB2003)*, volume 76 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In K. Aberer, V. Kalogeraki, and M. Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2003. ISBN 3-540-20968-9.

N. L. Tudor. Optimization of queries with conjunction of predicates. *International Journal of Computers, Communications and Control*, 2(3):288–298, July 2007.

W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11. USENIX Association, 1995.

C. Wang, B. A. Alqaralleh, B. B. Zhou, F. Brites, and A. Y. Zomaya. Self-organizing content distribution in a data indexed DHT network. In A. Montresor, A. Wierzbicki, and N. Shahmehri, editors, *Peer-to-Peer Computing*, pages 241–248. IEEE Computer Society, 2006. ISBN 0-7695-2679-9.

C. Wood. Playstation 3 clusters providing low-cost supercomputing to universities. *Government Technology*, 2011.

# List of Figures

## Thanks

Writing this thesis would have been impossible without encouragement and invaluable support from my advisors, colleagues, students and friends. I am indebted to all of you, and hope to have justified the use of your time. Therefore, the following list can only be incomplete. Nevertheless, I would explicitly like to thank the following people:

- Advisors Prof. Dr.-Ing. Robert Tolksdorf and Prof. Dr. Wolfgang Nejdl for their time and effort towards the completion of this thesis
- Prof. Johann-Christoph Freytag, Ph.D. and Dipl.-Inf. Ralf Heese for database-related advice and comments
- Dipl.-Inform. Tilman Liero for getting me out of several brain-freezes when dealing with operator tree reordering
- My parents, Conny and Rolf Mühleisen, for their unconditional support in so many ways – I would never have gotten to this point without you!

> "I will never forget the time when a disgusting gesture of history coincided with some desperate mechanism inside myself, and in six ~~weeks~~ months gave me ~~the book that altered my life~~ this thesis." – *adapted from John le Carré*

> "Things are only impossible until they're not!" – *Jean-Luc Picard*

# Deutschsprachige Zusammenfassung

Verteilte Speichersysteme stellen stets einen Kompromiss zwischen den Dimensionen Skalierbarkeit, Konsistenz und Verfügbarkeit dar. Insbesondere der Koodinationsmechanismus, mit dem Konsistenz zwischen den beteiligten Systemen hergestellt wird, ist für diesen Kompromiss maßgeblich. Neben dem Abruf einzelner Datenelemente wird auch die Bearbeitung komplexer Anfragen zur Unterstützung von Applikationen und Analysen zu einem zentralen Element dieser Systeme. Leider sind die Methoden zur verteilten Anfragebearbeitung bisher an einen konkreten Koodinationsmechanismus gebunden und lassen sich daher kaum an neue Systeme anpassen.

Aufgrund dieser Situation besteht die Herausforderung für diese Arbeit darin, die Bearbeitung komplexer Anfragen in verteilten Speichersystemen unabhängig von Netzwerkarchitektur und Koordinationsmechanismus zu untersuchen. Hierfür wurden daher möglichst umfassende Abstraktionen erstellt. Um zu zeigen, dass innerhalb der abstrahierten Umgebung die Ausführung komplexer deklarativer Anfragen effizient möglich ist, wurde eine ebensolche Methode erarbeitet. Hier bewegt sich der Prozess der Anfragebearbeitung durch das Netzwerk, während die Anfrage kontinuierlich optimiert wird. Eine theoretische Betrachtung dieses Prozesses ergab, dass ein logarithmisches Verhalten der Übertragungskosten im Bezug auf die Größe des verteilten Speichersystems möglich ist.

Um diese theoretischen Ergebnisse zu überprüfen, wurde zudem eine Reihe von kontrollierten Experimenten mit Hilfe von Simulationen durchgeführt. Um realistische Daten und komplexe Anfragen zu erhalten, wurde der Datenbank-Test TPC-H verwendet. Der Zusammenhang zwischen Parametern des abstrakten Koodinationsmechanismus und der Effizienz der Anfragebearbeitung wurde überprüft, wobei das vorhergesagte logarithmische Verhalten bestätigt wurde. Damit konnte gezeigt werden, dass bereits das vorgestellte minimale Netzwerkmodell bei Verwendung von nachbarschaftlicher Platzierung zusammenhängender Daten effiziente Anfragebearbeitung ermöglicht. Damit ist der Weg für die Implementierung komplexer Anfragen in einer Vielzahl verteilter Speichersysteme frei.

# Kurzlebenslauf Hannes Mühleisen

(Lebenslauf aus datenschutzrechtlichen Gründen entfernt)