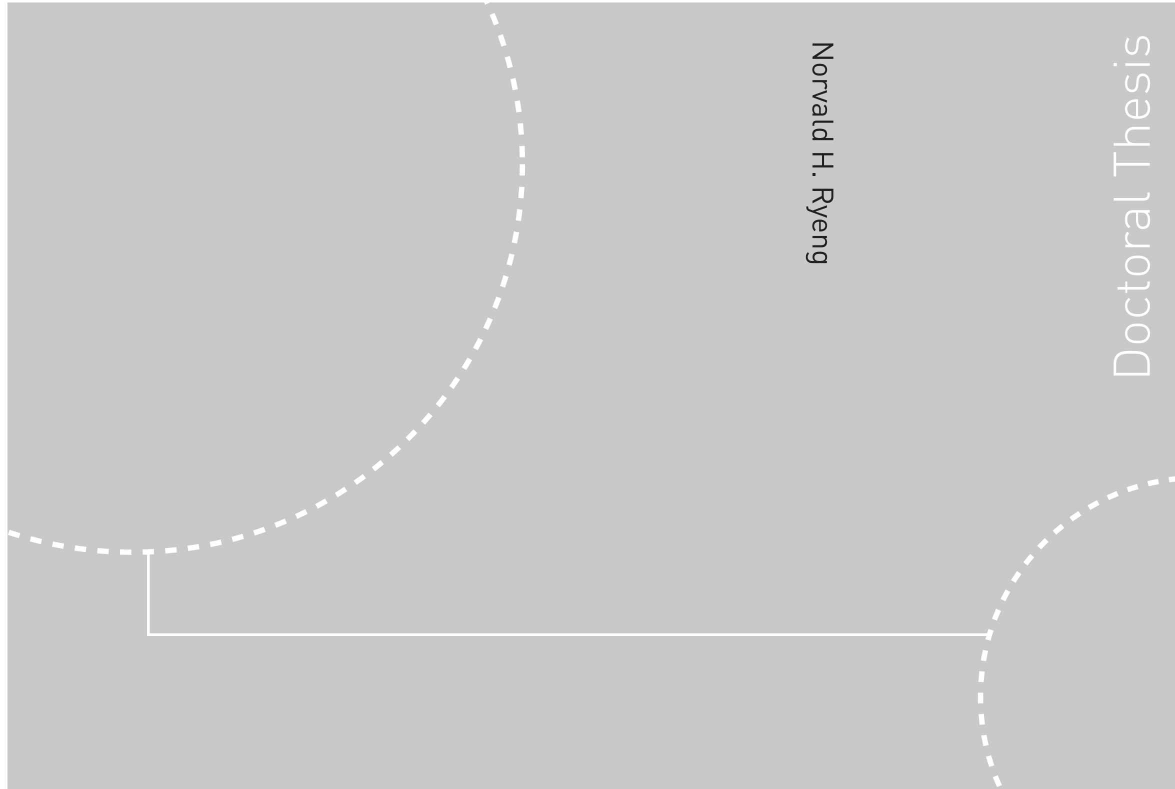
Doctoral theses at NTNU, 2011:293

Norvald H. Ryeng

Improving Query Processing Performance in Large Distributed Database Management Systems



ISBN 978-82-471-3158-9 (printed ver.)
ISBN 978-82-471-3159-6 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2011:293

NTNU
Norwegian University of
Science and Technology
Thesis for the degree of
Philosophiae Doctor
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Computer and Information Science

NTNU

Norvald H. Ryeng

Improving Query Processing Performance in Large Distributed Database Management Systems

Thesis for the degree of Philosophiae Doctor

Trondheim, November 2011

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

Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

©Norvald H. Ryeng

ISBN 978-82-471-3158-9 (printed ver.)
ISBN 978-82-471-3159-6 (electronic ver.)
ISSN 1503-8181

Doctoral Theses at NTNU,

Printed by Tapir Uttrykk

Abstract

The dream of computing power as readily available as the electricity in a wall socket is coming closer to reality with the arrival of grid and cloud computing. At the same time, databases grow to sizes beyond what can be efficiently managed by single server systems. There is a need for efficient distributed database management systems (DBMSs). Current distributed DBMSs are not built to scale to more than tens or hundreds of sites (i.e., nodes or computers). Users of grid and cloud computing expect not only almost infinite scalability, i.e., at least to thousands of sites, but also that the scale is adapted automatically to meet the demand, whether it increases or decreases. This is a challenge to current distributed DBMSs.

In this thesis, the focus is on how to improve performance of query processing in large distributed DBMSs where coordination between sites has been reduced in order to increase scalability. The challenge is for the sites to make decisions that are globally beneficial when their view of the complete system is limited. The main contributions of this thesis are methods to increase failure resilience of aggregation queries, adaptively place data on different sites and locate these sites afterwards, and cache intermediate results of query processing.

The study of failure resilience in aggregation queries presented in this thesis shows that different aggregation functions react differently to failures and that countermeasures must be adapted to each function. A low-cost method to increase accuracy is proposed.

The dynamic data placement method presented in this thesis allows data to be fragmented, allocated, and replicated to adapt to the current system configuration and workload. Fragments are split, coalesced, reallocated, and replicated during query processing to improve query processing performance by allowing more data to be accessed locally. The proposed lookup method uses range indexing to make it possible to efficiently identify the sites that store relevant data for a query with low overhead when data is updated.

During query execution, a number of intermediate results are produced, and this thesis proposes a method to cache these results and use them to answer other, similar queries. In particular, a caching method to improve execution times of top-k queries is presented.

Results of experiments in simulators and on an implementation in the DASCOSA-DB distributed DBMS prototype show that these methods lead to significant savings in query execution time.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Philosophiae Doctor at the Norwegian University of Science and Technology. The work described in this thesis was conducted at the Department of Computer and Information Science under the supervision of Professor Kjetil Nørvåg, with Professor Svein Erik Bratsberg and Dr. Olav Sandstå as co-supervisors.

Acknowledgments

First, I would like to thank my supervisor Professor Kjetil Nørvåg for his continuous guidance and his great interest and involvement as a supervisor. I would also like to thank Professor Svein Erik Bratsberg and Dr. Olav Sandstå for being co-supervisors.

I would like to thank Adjunct Associate Professor Jon Olav Hauglid for great cooperation over several years, and in particular for help with implementation in the DASCOSA-DB prototype. Dr. Christos Doulkeridis and Dr. Akrivi Vlachou make up an amazing research team, and I am very happy that we got to work together on a paper. Thank you. I would also like to thank the rest of my colleagues at the department for good discussions, inspiration and friendship.

I could never have completed this thesis if it had not been for the support of friends and family. They are too many for me to mention them all, but they still have my gratitude. I would particularly like to thank Jeanine Lilleng, Nina Knudsen and Dr. Hans Sverre Smalø for always being there when I needed it, from start to finish.

Contents

I Introduction **1**

1 Introduction **3**
- 1.1 Motivation 3
- 1.2 Research Focus 4
- 1.3 Methods 5
- 1.4 Thesis Outline 5

2 Background **7**
- 2.1 Peer-to-Peer Networks 7
- 2.2 Data Placement 13
- 2.3 Distributed Query Processing 17
- 2.4 Caching of Query Results 21
- 2.5 Distributed Data Storage and Query Processing Systems 22

3 Contributions **33**
- 3.1 Research Topics 33
- 3.2 Published Papers 35

4 Concluding Remarks **41**
- 4.1 Evaluation of Contributions 42
- 4.2 Future Work 43

Bibliography **45**

II Published Papers **53**

A Robust Aggregation in Peer-to-Peer Database Systems **55**
- A.1 Introduction 57
- A.2 Related Work 58
- A.3 Data Loss 59
- A.4 Fighting Data Loss 63
- A.5 Experiments 67
- A.6 Conclusion and Future Work 71
- Bibliography 72

B RIPPNET: Efficient Range Indexing in Peer-to-Peer Networks 75
B.1 Introduction . . . 77
B.2 Related Work . . . 78
B.3 Preliminaries . . . 79
B.4 Distributed Range Indexing . . . 81
B.5 Extensions . . . 85
B.6 Experimental Evaluation . . . 86
B.7 Conclusion . . . 90
Bibliography . . . 91

C Efficient and Robust Database Support for Data-Intensive Applications in Dynamic Environments 93
C.1 Introduction . . . 95
C.2 Background . . . 96
C.3 Overview of DASCOSA-DB . . . 97
C.4 Demonstration . . . 102
C.5 Future Work . . . 102
Bibliography . . . 103

D DYFRAM: Dynamic Fragmentation and Replica Management in Distributed Database Systems 105
D.1 Introduction . . . 107
D.2 Related Work . . . 109
D.3 Preliminaries . . . 111
D.4 Overview of DYFRAM . . . 115
D.5 Replica Access Statistics . . . 116
D.6 Fragmentation and Replication . . . 121
D.7 Evaluation . . . 126
D.8 Conclusions and Further Work . . . 136
Bibliography . . . 136

E Site-Autonomous Distributed Semantic Caching 139
E.1 Introduction . . . 141
E.2 Related Work . . . 143
E.3 Preliminaries . . . 143
E.4 Distributed Semantic Caching . . . 145
E.5 Experimental Evaluation . . . 152
E.6 Conclusion and Future Work . . . 154
Bibliography . . . 155

F Efficient Distributed Top-k Query Processing with Caching 157
F.1 Introduction . . . 159
F.2 Related Work . . . 160
F.3 Preliminaries . . . 162
F.4 ARTO Framework . . . 162
F.5 Answering Top-k Queries from Cache . . . 164

F.6 Remainder Queries 167
F.7 Server Selection 170
F.8 Experiments 170
F.9 Conclusion 172
Bibliography 173

G The DASCOSA-DB Grid Database System 175
G.1 Introduction 177
G.2 Overview of Related Systems 178
G.3 System Architecture 180
G.4 Distributed Data and Metadata Management 183
G.5 Distributed Query Processing in DASCOSA-DB 187
G.6 Distributed Monitoring and System Management 192
G.7 Experimental Evaluation 194
G.8 Summary and Future Challenges 195
Bibliography 196

List of Figures

2.1 Example of an unstructured network. 8
2.2 Example of a supernode network. 9
2.3 A Chord identifier circle. 11
2.4 Horizontal and vertical fragmentation of a table. 14
2.5 Four steps of distributed query processing. 18
2.6 Data and query shipping. 20
2.7 Caching opportunities. 22

A.1 Hierarchical aggregation using a DHT. 57
A.2 Variants of aggregation trees. 64
A.3 Accuracy of aggregation functions with different number of replicas. . 69
A.4 Accuracy of the count function with different node degrees. 69
A.5 Accuracy of the avg function with different node degrees. 70
A.6 Accuracy of the max function with different node degrees. 71

B.1 An example system. 79
B.2 Data distribution on a node. 82
B.3 Two-dimensional index of one-dimensional data. 83
B.4 Range placement. 85
B.5 Probability of index updates. 88
B.6 Messages used in varying network sizes. 89
B.7 Messages used when varying query widths. 90

C.1 High-level overview of the architecture of DASCOSA-DB. 97
C.2 Screenshot from the DASCOSA-DB system monitoring tool. 98
C.3 Query and performance under churn. 99
C.4 Example access pattern, and desired fragmentation and allocation. . . 100

D.1 Example access pattern, and desired fragmentation and allocation. . . 109
D.2 Dynamic fragmentation and allocation. 115
D.3 Histogram with four buckets and corresponding value ranges. 124
D.4 Results from two-site workload. 129
D.5 Comparative results from two-site workloads. 131
D.6 Results from workloads involving several sites. 133
D.7 Results from DASCOSA-DB implementation. 134

E.1 Example query from unmodified DASCOSA-DB. 144

E.2 Extended lookup message. 146
E.3 Extended lookup reply message. 147
E.4 Query dissemination with table fragment timestamps. 148
E.5 Result and timestamp propagation. 149
E.6 Execution time for varying network bandwidth. 152
E.7 Results for varying parameter distributions. 154

F.1 2D representation of query and data space. 163
F.2 Query plan transformation. 164
F.3 Cache containing the cache entries of two queries. 165
F.4 Areas examined by the remainder query vs. restarting a query Q_1. . . 168
F.5 Transferred data for 1,000 queries and uniform data distribution. . . . 171
F.6 Results of queries with varying k. 172
F.7 Results of queries with varying cache size. 173

G.1 Distributed architecture of DASCOSA-DB. 181
G.2 High-level overview of the architecture of a DASCOSA-DB site. . . . 182
G.3 Data fragmentation. 185
G.4 Query dissemination. 188
G.5 Query failure and restart. 191
G.6 Screenshot from the DASCOSA-DB system monitoring tool. 193
G.7 Execution time relative to baseline. 195

Part I

Introduction

Chapter 1

Introduction

Many databases have outgrown what single server systems can provide, and they continue to grow. To meet future demands for data storage, we need to build database management systems (DBMSs) that can scale. However, scaling is not easy. Distribution adds coordination overhead, and this overhead grows with the size of the system and limits scalability. A solution to the scalability problem is to make sites (i.e., the nodes or computers) in the distributed system more autonomous and thereby reducing the need for coordination, but this also has its downside: How can a less coordinated distributed DBMS process queries efficiently?

The advent of grid and cloud computing promises easier access to computing resources than ever, with the possibility to scale the resource allocation up and down as needed. The vision of distributed systems that automatically adapt resource usage to the users' needs seems to be within reach, but that requires distributed systems that can scale seamlessly from a single site to thousands of sites and adapt to the current workload [6, 97]. How can one build a DBMS that allows this type of automatic scalability, both in terms of storage and query processing?

The focus of this thesis is on how query processing can be done efficiently in a distributed DBMS with a large degree of site autonomy. This problem is addressed by looking at methods to improve data placement and query execution.

1.1 Motivation

In a large distributed system such as a DBMS in the cloud, one has to expect sites or network links to fail. A failing site takes with it the data stored on it, and a distributed system must be able to deal with this data loss. One strategy to deal with failures is to accept that some data will be unavailable and let queries return the best result that is possible given the currently available data. But how can query processing be made more resilient to failures? The current proposed solution for making aggregation queries resilient to failures is to replicate data and the whole aggregation process, in practice doubling the cost of each query. Because of this cost, it is not a viable solution.

With distributed systems comes the option of varying data placement. Assuming data can be placed freely in the distributed system, where should it be put? The

idea of a grid or cloud system is that the system scales with the load and storage requirements. This means that the workload and the number of sites available for data storage and query processing changes, and the system should adapt automatically to these changes.

There must also be an efficient way of locating these data, even as they are moving around to match the current workload. With a large number of changes, it is important that this lookup mechanism does not need to be updated every time a data item is changed. Current structured networks mostly store tuple indices that must be updated every time a tuple changes.

Current DBMSs cache data as it is read from disk, and some also cache the final results as they are delivered to the user. However, they do not cache intermediate results, i.e., the results of subtrees of the whole relational algebra tree for the query. A query is specialized to return only the relevant tuples and attributes, while the intermediate results are more general and may be used to answer more queries. Hence, there should be more to gain from caching these intermediate results than only final results.

1.2 Research Focus

The topic of this thesis is how to improve the performance of query processing in large distributed DBMSs where autonomy has been increased and coordination has been reduced in order to improve scalability. The reduced coordination costs is beneficial to scalability, but it comes at the cost of less information available at each site. Each site decides autonomously, based only on the locally available information, how to process queries. The challenge is to do this in a way that is beneficial to the system as a whole.

In particular, this thesis focuses on data distribution and lookup, caching of intermediate results and how to deal with site failures during processing of aggregation queries.

1.2.1 Research Questions

The main research question for this thesis is:

> *How can query processing in large distributed database management systems be made more efficient?*

This question covers too large an area of research to fit into one thesis, so it has been further specified into the following questions:

1. How are aggregation operations influenced by site failures?
2. How can data placement adapt to constantly changing workloads?
3. How can data be located if sites are given autonomy over storage decisions?
4. How can caching of intermediate results be used to speed up query processing?

These questions will be revisited in Chapter 4, where they will form the basis against which the contributions of this thesis will be evaluated.

1.3 Methods

The main contribution of this thesis is methods for failure resilient aggregation, data placement and lookup, and caching of intermediate results. These methods have been evaluated by doing simulations and experiments on implementation in a distributed DBMS prototype.

There are two main advantages of simulators: control and scale. By simulating only enough of a system to conduct the necessary experiments, it is easier to have full control over the parameters and environment of the simulations. A real system is influenced by external events, while a simulator creates a fully controlled environment where external events have no effect on the simulation. By limiting the details or depth of the simulation, it can be scaled to cover a large number of sites, thereby offering an opportunity to examine larger systems than otherwise possible.

The main limitation of simulations is reduced realism since the simulator is not a real system in a real environment. For the results to be relevant, it is important that external events that will have a significant effect on the system in a real implementation are also modelled in the simulator. It is easier to measure effects in a simulator, but fewer metrics are available since the metrics are limited by the simulator model.

Implementation in an existing system or prototype offers a realistic setting, but it is harder to control the environment and more costly to scale. The effect of external events on the system is both a strength and a weakness. It is a strength because it is the effect of a real environment, but it is also a weakness because it makes it harder to understand what is actually the cause of the effects that are measured.

Experiments in existing systems or prototypes are also harder to scale than simulations. Since the system is real, and not only simulated, real hardware is needed to run experiments. Scaling experiments to thousands of sites is possible in simulators. Finding thousands of networked computers to run a prototype DBMS is harder.

For the work presented in this thesis, a combination of simulators and implementation in prototypes are used. Some methods are evaluated only in simulator or prototype, while other are evaluated using both. The DASCOSA-DB distributed DBMS prototype has been developed concurrently with the research included in this thesis and has been used to evaluate several of the proposed methods. Work conducted prior to the development of the DASCOSA-DB prototype has been evaluated using simulators.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 contains background information on peer-to-peer systems, data placement, query processing and caching.

It also contains a short review of relevant distributed storage and querying systems. Chapter 3 describes the papers included in this theses and how they fit together. Chapter 4 concludes the thesis.

The published papers are included, in chronological order, as Part II. These papers are reproduced faithfully with regard to the published text, except for corrections to minor typographical errors. To increase readability, the papers have also been reformatted to fit the thesis format. This reformatting has not altered the contents of the papers.

Chapter 2

Background

This chapter contains background material that will serve as an introduction to the concepts that are necessary to understand the papers included in this thesis. The background material has been divided into five areas: Section 2.1 is an introduction to peer-to-peer systems and overlay networks, in particular structured overlay networks. Section 2.2 discusses the problem of how to split up database tables and allocate the fragments to sites in a distributed DBMS. Section 2.3 explains distributed query processing. Section 2.4 describes how caching is used in query processing in distributed DBMSs. Finally, Section 2.5 presents some existing systems for distributed data storage and querying.

2.1 Peer-to-Peer Networks

Large-scale distribution is hard. Large systems consist of a large number of computers, and if the probability of a single computer failing remains constant, the probability of a single failure in the system increases with the number of computers. If the system is large enough, failures are to be expected as part of normal operation. This means that systems must be constructed in such a way that single failures do not bring down the whole system. It is especially important to avoid creating any single points of failure.

One of the solutions that have been proposed is the use of peer-to-peer overlay networks. Peer-to-peer overlay networks remove the need for a centralized server that easily becomes a bottleneck and single point of failure. This allows the peer-to-peer networks to grow to considerable sizes [4].

The overlay networks can be broadly categorized into three different categories: unstructured networks, supernode networks, and structured networks. The structure of the overlay network decides the fundamental principles of communication and hence affects the capabilities and limitations of the systems built on top of the overlay network.

In general, we describe an overlay network as a graph $G = (V, E)$ with a set V of sites and a set E of network links between these sites. The different categories of overlay networks are mostly distinguished by restrictions on which network links may exist in E, but some networks also distinguish between different types of sites.

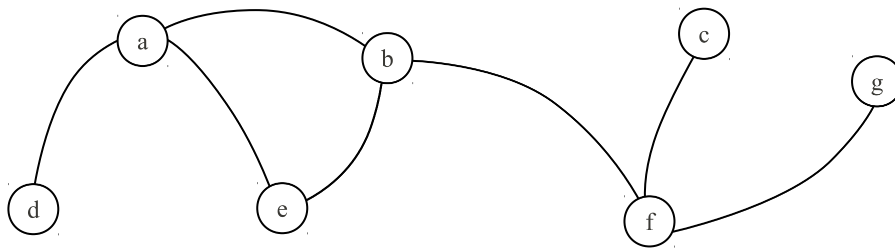


Figure 2.1: Example of an unstructured network.

2.1.1 Unstructured Networks

Unstructured networks are networks where any site can connect to any other site without restrictions. A site may connect to several other sites, so that it is not disconnected if one network link or site fails. The extra connections also allow for direct communication with more sites. An example of an unstructured network is shown in Figure 2.1. In this example, site e will stay connected to sites a and d even if site b disappears. However, the failure of site b will partition the network into two parts. In order for the network as a whole to survive the failure of site b, more connections have to be made between the two partitions.

The Gnutella file sharing system [42] and Freenet [26] are examples of systems using unstructured overlay networks.

Storage and Querying

A key feature of unstructured networks is that there are no restrictions on data placement. All sites have full autonomy over which data items they store and which other sites they connect with. This means that each site can store those data items it currently needs and rely on the network to get new ones if necessary.

The disadvantage of this approach is that a query has to be broadcast to all sites in order to retrieve all possible data items, as there is no structure that can provide more efficient lookup mechanisms. If site e in Figure 2.1 requests some data, a query will be sent out to sites a and b, which again will forward it to sites d and f, respectively. Site f will forward the query to sites c and g. In large networks, this broadcast scheme is infeasible, and a time-to-live field is inserted into queries to limit the number of routing hops a query may travel before it is dropped. This limited horizon effectively limits what data items are visible to a site and means that it is impossible to give lookup guarantees in unstructured networks.

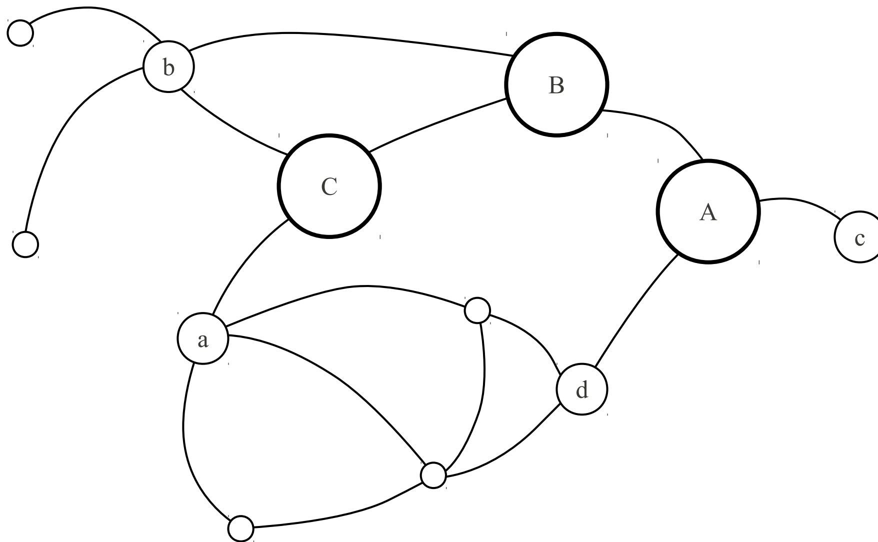


Figure 2.2: Example of a supernode network. Capital letters represent top-level sites, lower case letters represent mid-level sites, while unnamed sites are on the lowest tier.

2.1.2 Supernode Networks

Supernode networks are much like unstructured networks, but some sites have been appointed supernodes and given a special status in the system. There may be multiple levels of supernodes, forming a hierarchy. Sites connect to other sites in the same or the next tier, with the sites of the highest tiers forming the core of the network. That is, the network $G = (V, E)$ consist of a set $V = V_1 \cup V_2 \cup V_3 \cup \cdots \cup V_T$ of sites in T tiers (1 being the highest tier). Each site v is member of only one tier V_t. A network link $e = (v_1, v_2)$ from site $v_1 \in V_t$ to v_2 is allowed only if $v_2 \in V_{t-1} \cup V_t$.

Sites with slow network links or low processing power are usually placed in the lowest tier, with more powerful and well-connected sites as supernodes higher up in the hierarchy. This organization helps to avoid slow network links or sites becoming bottlenecks in the system. Figure 2.2 shows a network with three tiers, where sites A, B and C are in the top tier and sites a, b, c and d are in the middle tier. All unnamed sites are in the lowest tier.

FastTrack [102] and Napster [71, 72] are examples of systems using supernode overlay networks.

Storage and Querying

Supernode networks are more organized than unstructured networks. The supernodes play an important part in this organization, and are often given the task of indexing data stored in the network. The autonomy of the unstructured network is

retained, allowing them to store the data items they need.

Queries for new data items are sent to the supernodes. The supernodes have indices of the data items on sites connected to them and may ask other supernodes about items from their subtrees of the network. The list of sites that store relevant data items is sent back to the querying site and this site may connect directly with any of those sites to retrieve the item. This hierarchical index lookup makes querying large networks feasible. Traditionally, peer-to-peer networks have been designed mostly for file sharing, but effort has been made to construct routing indices that enable efficient query processing [36].

2.1.3 Structured Networks

Structured networks impose a structure on the network, restricting which connections may exist between the sites. There is a large variety of possible structures, but two have emerged that have proven very useful for database applications: distributed hash tables (DHTs) and tree structured networks.

Hash tables are well known for their efficient lookup properties, and distributed hash tables deliver some of the same performance to overlay routing. Each site is given a unique key, usually by hashing on some identifying data, e.g., network address. All data items stored in the system are also given addresses in the same address space by hashing on a key, and each site is given responsibility for storing data in a range of the data space. Examples of DHTs are Kademlia [64], Chord [92], CAN [83], Pastry [87], Tapestry [48] and Bamboo [86]. Common to most of these is that they have a simple ring-like structure, as shown in Figure 2.3. Each site in the ring is given the task of storing data items that fall between this site and its closest neighbor with a lower address, i.e., data items are stored on the successor site of the item's position in the ring. Messages are routed from site to site in jumps along the ring according to routing tables held by each site.

A structured network can also take on a tree structure. This class of structured networks include systems such as the Distributed Segment Tree [113], the Range Search Tree [39], BATON [54], P-Grid [2] and P-Tree [30]. A similar approach is search tries stored in DHTs [106].

Storage and Querying

The strength of structured networks is the ability to directly address data items and route messages to the site responsible for the address space to which the data item belongs. The disadvantage of this approach is that it forces a data partitioning on the sites in order to balance storage and querying load, making the structure of the network decide how data is partitioned. Often, the data is spread out uniformly using a hashing function to avoid any sites becoming hotspots and possible bottlenecks in the system. Despite the forced data placement, structured overlay networks have their advantages. Efficient routing algorithms and guaranteed data discovery make structured overlay networks very popular for database applications.

The DHTs only support exact match lookups. More advanced queries, such as range and cover queries (i.e., queries for all data items in a given range or all ranges

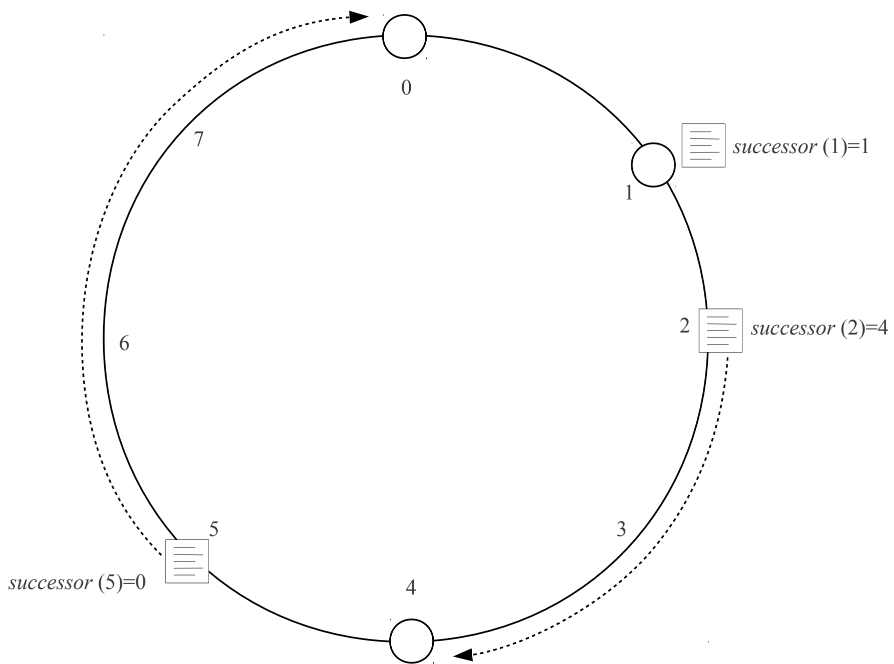


Figure 2.3: A Chord identifier circle with eight possible addresses. The system consists of three sites with addresses that hash to the values 0, 1, and 4. Three data items with identifiers that hash to 1, 2, and 5 are stored in the network.

that cover a given data point), are made impossible by the same hashing function that gives DHTs their advantage. To do a range query in a DHT, one would have to look up every possible key in the range, creating an excessive amount of lookup requests, especially if the data space is sparsely populated. In [112], two methods for doing range queries in peer-to-peer systems are discussed. The first is simply to give the same hash key to all values within a range. This would reduce the number of hash keys used and hence the number of lookup requests needed, with the disadvantage of clustering data on those sites. Another drawback of this method is that it only supports predefined ranges. The other method suggested is to create a multicast group for each range. Queries are then multicast to the ranges that are relevant.

A method for range selection in DHTs based on locality sensitive hashing is presented in [43]. The method can look up a range in $O(\log N)$ hops in an N-site network, but it only gives an approximate answer. Another approach is to use locality-preserving hashing functions to place similar values close to each other in the hash space, as is done in HotRoD [79], which combines the locality-preserving hashing with replication to support range queries. Another method is to use Gray coding, as done by GChord [114], so that consecutive values differ in only one bit. This bit change is used to forward the range query through a Chord network, reaching all data points within a range.

The simplest method is perhaps to skip the hashing step altogether, as is done in [1] and [14]. By skipping the hashing step, contiguous values are placed after each other in a ring structure that is otherwise the same as in a DHT-ring. This has the obvious disadvantage of being unbalanced if the data set is skewed, but these methods add load balancing algorithms to ensure fairness.

Tree structures are often applied in database systems, and they support range queries. Distributed segment trees are binary trees that support both range and cover queries, but requires updates to sites in all levels of the tree every time a tuple is inserted or deleted. Techniques exist to reduce the load on the root and sites close to the root somewhat, but still a message has to be sent to these sites [113]. This creates a lot of messages every time a tuple is inserted or deleted and limits scalability.

P-trees [30] are built on top of a Chord [92] DHT where each site stores part of a structure similar to a B^+-tree. Tuples are stored in leaf sites, and these sites constitute the Chord ring, similar to the linked list of leaves in a B^+-tree. However, P-trees with a large number of children at each level are not very efficient for frequently updated databases because of high maintenance costs. Another tree structure, P-Grid [2], uses a binary prefix tree. The disadvantage of P-Grid is that it in case of skewed data may degenerate into a linked list, which severely limits scalability [54]. The BATON [54] tree structure uses self-adjustment to avoid these problems in case of data skew.

A number of range indexing techniques are built on skip lists. Skip lists are trees of linked lists where the lowest level is a linked list of all sites. Lists higher up in the tree skip a certain number of elements ahead, creating increasingly sparser lists for higher levels. Skip lists are used by SkipIndex [111], SkipNet [46], Skip Graphs [13]

and ZNet [90].

Unstructured and supernode networks do not have the same problem with range queries. A range query in an unstructured network can be broadcast in the same manner as other queries, and in supernode networks, the supernodes can look up ranges in their indices. DHTs are built for direct lookup, and struggle with range queries because of the coupling between querying mechanism and data placement. Queries are formulated as direct data item lookups, and the data item is located by message routing. The structure of the network forces data placement restrictions on the sites, typically using hashing functions to achieve load balancing, making it hard to keep ranges together. Combined with direct tuple addressing, this makes range queries hard.

2.2 Data Placement

In a distributed DBMS, there is a question of where to store data. Data can be stored centrally on one site, split up and distributed to a set of sites, stored in its entirety on all sites, or any solution in between. The options are infinite. It is hard to know in advance which configuration is the best for a given system.

To improve query processing performance, a data placement algorithm should aim to place data so that they can be accessed efficiently. If data is stored on other sites than where they are used, they have to be accessed remotely. Usually, remote accesses are much slower than local accesses since the data has to be fetched over a network. The optimal would then be to store data such that all accesses are local, but unfortunately that is not always possible. Also, we expect failures to occur, which means that data must be replicated to survive site failures. This will also increase the number of sites that can access data locally, but it incurs some extra cost in keeping the replicas up-to-date.

2.2.1 Fragmentation

The problem of data placement can be divided into three subproblems: fragmentation, fragment allocation, and replication. Fragmentation is about dividing a table into a set of table fragments. Figure 2.4 shows two ways to fragment a table. Fragmentation can be done horizontally, splitting the table between the rows so that all attributes of a tuple are stored together in the same fragment. It can also be done vertically, splitting the table between columns. With the latter method, all fragments would contain the key attributes and one or more other attributes for all tuples. This is known as a column store.

The horizontal fragmentation shown in Figure 2.4 is a fragmentation into ranges, where each fragment consists of a continuous range of rows. It is also possible for fragments to contain a set of single rows from many places in the table. An example of a fragmentation method that would result in such fragments is horizontal fragmentation based on the hash of key attributes. This type of fragmentation occurs if, e.g., a DHT is used to store database tuples. The DHT places a data item on a site based on the hash of the item's key, and all items with keys that hashes to

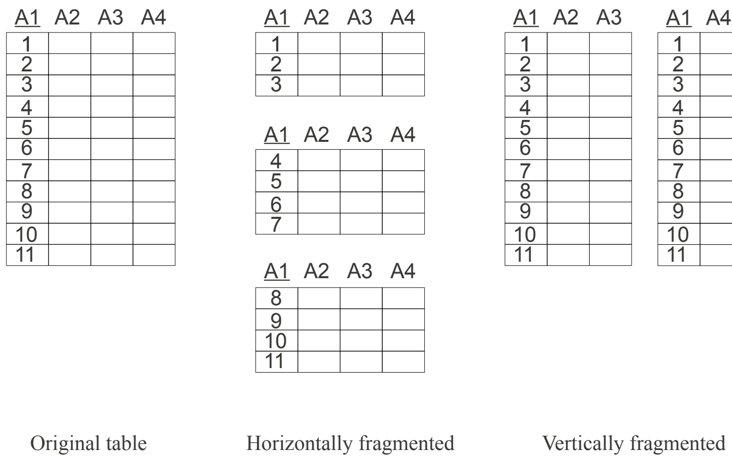


Figure 2.4: Horizontal and vertical fragmentation of a table. Attribute A1 is the key.

a value in the same address range (i.e., the range of the DHT address space that a site is responsible for) belong to the same fragment.

The fragmentation problem is particularly visible when multiple sites contend for access to data in the same region. How should the data be fragmented to minimize contention? If two sites both access data in the same region, the data could be split into two fragments with either the contended data in one of the regions or split so that each fragment contains a part of the contended region. A third option is to split the region into three parts, two accessed exclusively by either site and a third that contains the contended region.

The problem of fragmenting tables so that data is accessed locally most of the time has been studied throughly [8, 27, 35, 49, 53, 82, 88, 89, 91, 95, 104, 115]. It is also related to some of the research in data placement in distributed file systems (see a summary in [41]). One of the important differences between distributed file systems and database systems is that of granularity. Whereas file systems address files and disk blocks, database systems address tables and tuples. Distributed database systems also need a fragmentation attribute that can be used to define partitioning rules for the table.

2.2.2 Fragment Allocation

After fragmentation, the problem of fragment allocation arises. Given a set of fragments, which fragment should be allocated to which site? The basic principle is to allocate the fragments in such a way that most accesses are local. This is not as easy as it may sound since, in general, there are more than one site competing for access to the same data. If local access is not possible, which remote site is most

beneficial?

In case of multiple sites accessing the same fragment, allocation algorithms have to decide if the fragment should be allocated to one of the sites accessing it, perhaps the site that has the most accesses to the fragment, or if the fragment should be placed on a separate site, where all contending sites have equal access to it.

Fragmentation and fragment allocation are tightly coupled. There are methods that do only fragmentation [8, 82, 89, 104, 115] and methods that do only allocation of predefined fragments [9, 12, 18, 28, 38, 65, 98]. Some methods also exist that integrate both tasks [27, 35, 49, 53, 88, 91, 95].

The intended lookup strategy also influences data allocation. As described in Section 2.1, data placement strategies and lookup strategies are tightly connected. When looking up data, a clear structure is beneficial since it allows direct access to interesting data items. However, data usage is usually not that structured, which means that fragment allocation and lookups are often opposites that must be balanced according to the expected use of the system. E.g, a DHT forces a certain fragmentation and allocation in order to achieve guaranteed lookup, while an unstructured overlay network sacrifices lookup guarantees in order to achieve a free fragment allocation.

2.2.3 Replication

A third question in data placement is replication. How many replicas should there be of a single table fragment? Replication is often thought of in connection with failure resilience since multiple replicas make the system more redundant to site failure, but redundancy is not the only advantage. Replication can solve some of the problems with multiple sites contending for the same fragment by allocating the fragment to all of them. This way, remote accesses are converted to local accesses. This is an advantage as long as there are many read operations, but write operations become more costly as there are now multiple replicas to update. This also complicates write operations as there might be consistency requirements that require updates to be executed simultaneously in all replicas.

While fragmentation and fragment allocation are typically integrated, replication has usually been studied separately [15, 25, 45, 67, 68, 103]. Still, there are some methods that take an integral view of fragmentation, allocation and replication [35, 91, 95]. Some of the replication strategies allow for dynamic replication [15, 45, 67, 68, 103], creating new replicas when needed. However, one must be careful so that the cost of dynamic replication does not exceed the alternative of remote reads and writes [25].

2.2.4 Static Data Placement

Fragmentation, allocation and replication methods can also be categorized as either static or dynamic. Static methods analyze an expected database workload and produces an optimized static data placement. The workload used as a basis for static methods is typically a set of database queries gathered over time from the live

system, but it can also include inserts and updates. It can be argued that static methods could gain from looking at a sequence of operations instead of a set [7], but sets are still the norm.

Some methods also utilize more particular information on the data in addition to the query set [89]. The disadvantage of this information, is that it has to be entered manually by a user with knowledge of the data. It is not possible to get this kind of information automatically. The design advisor [115] is one type of static method that bypasses the problem of gathering additional information by delivering its decisions as suggestions of possible actions to a human database administrator. The database administrator would then use his knowledge of the data to adapt the advisor's suggestion.

Common to static methods, are that they are offline methods, used only at major database reconfigurations. They do not adapt to changes in the workload. Some approaches, such as evolutionary algorithms for fragment allocation [9, 28], lend themselves easily to the static setting, since it is possible to give them time to run through many generations on a non-changing workload.

2.2.5 Dynamic Data Placement

A static method precomputes fragmentation, allocation and replication. However, this optimal configuration may quickly become suboptimal if the workload changes. Sites may change hotspots, or the ratio of reads to writes may change. Such variations can take place over days, e.g., changing from a working day to a holiday workload, or it may be faster changes measured in minutes or seconds. Static methods cannot handle these types of workload changes.

Dynamic methods continuously monitor the database to adapt fragmentation, allocation and replication to the changing workload. These methods are part of the trend towards fully automatic tuning of database management systems [101]. There have also been efforts to integrate vertical and horizontal partitioning while also taking other physical design features like indices and materialized views into consideration [8].

Research in adaptive data placement has focused mostly on load balancing, either by data balancing [27, 49] or by query analysis [53]. Data balancing tries to keep the amount of data similar across sites, while query based balancing tries to keep the computational load similar. The results are not always the same. If there are hotspots in the data accesses, the sites storing these hotspots will have a greater load per data item than other sites in the system. In this case, the two approaches to load balancing will give different results.

While load balancing has been a popular field of study, the idea of continuously moving data to optimize access has received less attention. The work of Brunstrom et al. [18] is an exception to this. In their system, predefined fragments are periodically considered for reallocation based on the number of accesses to the fragments. This work, however, does not integrate fragmentation and replication.

Unlike static methods, dynamic methods have to associate a cost with changing the existing fragmentation, allocation or replication. Refragmentation is cheap if

fragments are only split, but if fragments are to be coalesced, some of them might have to be moved to other sites, incurring data transportation costs. The same applies to fragment allocation. Replication has a cost model where it is cheap to remove a replica since all that is necessary is to delete it locally. Creating a new replica, however, is expensive since the whole fragment has to be copied to a new site.

An additional challenge is that it is hard to predict how the workload will be in the future. Adaptive methods must base their decisions on historical information, but the workload might change so that the new configuration does not pay off before the workload changes again. This has the effect that some adaptive changes may worsen the situation instead of improving it. The algorithms should try to avoid adapting too quickly to changes, but must also not change too slowly.

2.2.6 Distributed Decision Making

Another aspect of data placement methods is distribution of control. Some methods use a dedicated centralized server for gathering information and decision making. Other methods employ a completely distributed architecture, allowing each server to make autonomous decisions. Among the latter class of systems, we find replication schemes for mobile ad hoc networks (see [78] for an overview). However, these approaches do not consider table fragmentation and in general do replication decisions on a quite coarse granularity. A combination of the centralized and decentralized methods also exist. The servers are organized in smaller groups, and each of these groups chooses its own coordinator that makes decisions for the whole group, thus behaving like a centralized coordinator for the group [45, 67].

An interesting model for adaptive, decentralized data placement, is that proposed for Mariposa [91, 94], which uses an economic model with a bidding system to adapt to changing workloads. A Mariposa site will sell its data to the highest bidder in a bidding process. Queries will buy data to have them locally accessible instead of settling for remote accesses and high access times, but have limited budgets. Mariposa's method is dynamic, adapting to the situation at the time of query planning, optimizing for queries with large budgets.

2.3 Distributed Query Processing

This section covers mostly general distributed query processing methods. A more in-depth description of such methods can be found in [77].

A query posed to a distributed system is assigned a coordinator site, e.g., the site that was contacted in order to pose the query. This could be the same site for all queries, or the coordinator role could be distributed so that each query has its own coordinator. The coordinator processes the query by going through four basic steps, shown in Figure 2.5.

The coordinator starts by decomposing the query from the form it was posed by the user, typically SQL, to algebraic form that is used internally by the system. After decomposition comes the data localization step where the coordinator site

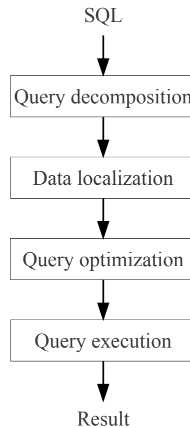


Figure 2.5: Four steps of distributed query processing.

locates the data fragments necessary to process the query. The result is a basic query plan that explains where to get data and how to compute the final result. This query plan is improved by the query optimizer step. The optimizer chooses a strategy for executing the query, including join order and how data fragments are scanned. Finally, the optimized query is executed and a result is produced and returned to the user.

Query decomposition is a translation from one query language to the internal query language of the DBMS and is mostly the same for both centralized and distributed systems, but the localization, optimization and processing steps deserve a closer examination. In a real system, these steps are not necessarily separate but may integrate and overlap. However, it is easiest to examine them one by one.

2.3.1 Data Localization

Data localization is nonexistent in a centralized system since all data needed by the query are located on a single site. In a distributed system, these data have to be found. The query describes which tables are used, and in some cases also a range of values for an attribute in that table. Using this information, the localizer must find which sites store relevant data.

Systems that store tuples directly in a DHT have rules for data placement and know that all sites potentially have relevant data. Data items are retrieved one by one and there is no need to look up which sites are involved in advance. However, if sites are allowed to store data independently of a DHT, a global catalog is needed. The global catalog stores information on where data is located, i.e., a mapping between tables or data ranges of tables and sites. This catalog may be centralized or distributed.

In a large distributed DBMS, an important part of the localization step is to rule out sites from the query. If sites are not ruled out, each query has to be broadcast

to all sites, leading to inefficient use of the system. The localization step limits the involved sites to those that are actually needed to process the query. The result of localization is a query plan that describes how and where to get and process data in order to get the final result.

2.3.2 Query Optimization

Query optimization makes a series of improvements to the query plan. A significant optimization is to find a good join order, since this often has a great impact on query execution time. Other optimizations involve access methods. The localization step has found relevant primary and secondary indices, and the optimizer may choose one of these or fall back on a complete scan of a table or a table fragment. These optimizations are not separate, so the existence of an index may affect join order.

The optimizer also has the option of choosing replicas. It could choose the replica assumed to be the fastest to access, or it may choose to access multiple replicas in parallel to speed up access.

A survey of query optimization can be found in [52].

2.3.3 Query Execution

There are two basic strategies for query execution in a distributed system: data shipping and query shipping. Figure 2.6 shows how execution is distributed for both strategies for a query joining three tables, T, U and V. Table T is split into two fragments, T_1 and T_2, while the other two tables consist of one fragment each. When using data shipping, all query execution takes place on a single site. This may be the same site for all queries, i.e., a central query execution site, but it can also be a different site for each query. In any case, there is only one site executing a single query. Other sites that are involved are used as data storage sites that only execute read and write operations.

The data shipping strategy moves data to the site of query execution. A query shipping system, on the other hand, moves the query to the sites that store the data. If the result of one operator is used by an operator on another site, the site of the first operator will ship an intermediate result to the next site. In the end, the last operator produces the final result that is sent back to the user.

The optimizer decides which algebra operations are to be executed at which sites. One strategy is to reduce the amount of data transferred between sites, which usually means that as many operations as possible are done locally where the data are stored. If data from two sites are used by an operator, the operator is located at the site that has the largest operand, resulting in only the smallest operand being sent over the network.

Query processing in large distributed systems present new problems to data owners. One is the issue of trusting a cloud provided site to execute queries correctly. Data owners can cryptographically sign data to make sure changes are discovered, but results of query operators are harder to verify [108]. Also, there is a question of privacy. Not all data can be shared with the cloud provider, and efficient execution

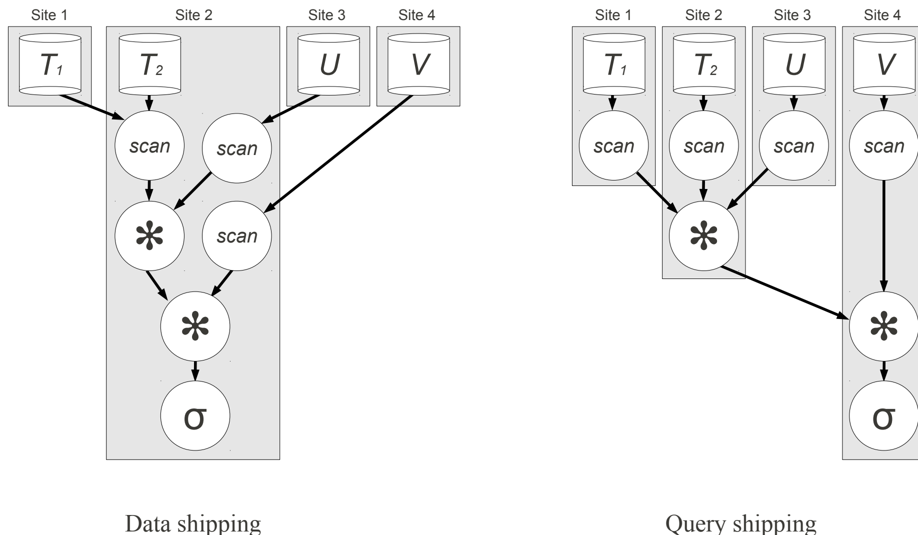


Figure 2.6: Data and query shipping.

of privacy sensitive queries is a current research topic [5].

Distributed Operators

Figure 2.6 shows a query shipping system where each relational algebra operator is processed on a single site. It is also possible to distribute the processing of one operator. One example of this is the use of a reduction tree to process aggregation queries, as is done in, e.g., TAG [63]. A tree of sites is constructed, and each node in the tree produces a partial aggregate covering its own data. Internal nodes in the tree combine their own partial aggregate with those of their children, propagating partial aggregates from the leaf nodes towards the root. The root produces the final aggregated value covering all sites.

Another example of distributed operators is the equijoin operator as implemented in PIER [51]. The hash based redistribution join exploits the hashing function of the underlying DHT to do join attribute matching. Both operands are inserted into the DHT using the join attribute as key. This results in tuples with matching keys from both operands to be stored on the same site. Each site then joins the tuples they have received and sends the result back to the coordinator. This processing is similar to a MapReduce [32] using the hashing function to map and join to reduce.

MapReduce can also be used as a framework to execute queries [40, 96]. Queries are decomposed to a representation of specialized map and reduce operations that are then executed in a cloud running, e.g., Hadoop. Traditional query execution and MapReduce are useful for different types of queries [93], so systems have been built that combine techniques from parallel DBMSs and MapReduce and select the best method for each query [81, 105].

2.4 Caching of Query Results

The principle of locality states that accesses are clustered so that a process that accesses a particular data item is more likely to access the same item again or to access items in its close proximity than it is to access other data items. This principle can be applied to a single query, a transaction, a site or to the complete distributed query workload. This principle does not always hold, but when it holds, it makes it possible to save time by caching data.

DBMSs typically cache disk blocks to speed up disk access. A data shipping distributed DBMS may also cache data that is received from other sites. On a higher level, it is possible to cache parts of tables [11, 16, 62]. The coordinator may have ordered an indexed scan on a table fragment on another site and may cache the contents it receives. This is still caching of the raw data that is needed to process queries, but on a higher level than disk block caching. The results are readily available if the same operation is requested at a later time.

The use of a result cache that caches the final result of a query has been implemented into MySQL [70]. The final result of SELECT queries are cached, and if the exact same query reoccurs, it is answered from cache if this is still valid. The query has to be exactly the same in SQL form, byte by byte, for the cache to be used. The result cache gives good results in systems where the clients frequently pose the same query, e.g., a web site that shows the last ten entries on its main screen. A similar caching is available in Oracle 11g [60], but the caching is explicitly requested by the client.

Unlike the result caches, semantic caching [31] and predicate-based caching [56] are two methods that try to exploit cached results also when the new query does not completely match the cached query. These methods add semantic descriptions to cached data and later use these to match new queries to cache entries. This means that more queries can reuse the already cached result.

When a query is posed in such a system, the cache is checked for semantically described cache entries that are similar to the new query. If such an entry is found, the query is split into two queries: one query that reads from cache and a remainder query that builds on the cached data to construct the result originally requested by the new query. If the cached entry contains more information than is needed by the new query, it is narrowed down. However, if the cache entry is not enough to answer the new query, the remainder query must fetch this data and merge it with the cache entry. This may include joining the cached result with a new table, extending ranges of WHERE clauses and other limitations. The system must take care so that the remainder query and the merge of the remainder query result with the cache entry is not more costly than the new query originally was.

If the construction of a remainder query is successful, the semantic caching produces little overhead and reduces network traffic in the distributed DBMS [55, 84]. Semantic caching has also been applied to deductive databases [21] and web querying systems [24, 61].

The caching of input data and final results concerns data at each end of the query, but less attention has been given to caching opportunities in stages between

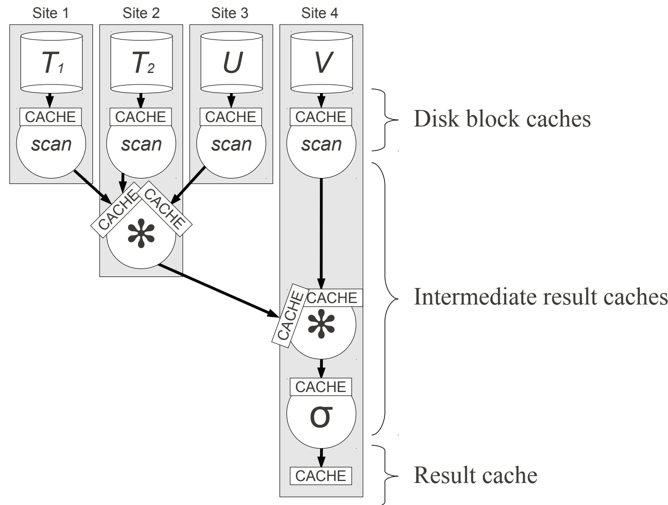


Figure 2.7: Caching opportunities.

input and output. MySQL caches the result of SELECT queries, but does not cache the results of nested queries. In a query shipping system, intermediate results are shipped between sites and therefore readily available for caching. Figure 2.7 shows the different caching opportunities in query processing. In the figure, only the receiver caches data. It is also possible to cache data on the sender side.

Usually, a cache caches data that happens to pass by it. Queries are cached automatically with the expectation that they will be reissued. With cache investment [57], the DBMS deliberately executes suboptimal queries in order to generate cache entries that are expected to have a high hit rate. This means that one query is deliberately slowed down in order to improve the execution of (expected) subsequent queries. This cache investment expects to amortize the costs of a suboptimal query execution over time as more queries are able to use the cached data than otherwise would have been possible.

Taken further, a DBMS may use view materialization [23, 66] to always keep the results of some interesting queries precomputed. These views are usually defined manually by the database administrator.

2.5 Distributed Data Storage and Query Processing Systems

A number of systems have been constructed for distributed data storage and query processing. Some systems only provide data storage, other provide only query processing, and some provide a combination of these. Some are built for a specific

purpose, while other are more general storage or query systems.

The disadvantage of caching and materialization is that data must be kept up-to-date or invalidated when the base tables are changed in such a way that the cache or materialized result is affected. If the update strategy is chosen, this adds a cost to update requests. If invalidation is chosen, it adds a cost either to updates or to querying. Delays in updates and invalidation can be accepted if consistency is not strictly enforced.

2.5.1 Data Storage Systems

Data storage systems provide persistent storage for data using various data models. Common to these systems is that the querying capabilities are minimal. Such facilities are expected to be provided by systems that build on the storage systems. A survey of scalable data stores can be found in [20].

OceanStore

The OceanStore system [59] is designed to provide persistent storage on a global scale. A wide variety of consistency guarantees, ranging from simple file system access to ACID-type transactions, is possible. It also supports replication to improve availability and access time.

Data access is by globally unique identifiers (GUIDs). Names are converted to GUIDs by reading a catalog object. The lookup process is bootstrapped by looking up a well known GUID for the root of the catalog hierarchy.

OceanStore is based on an unstructured peer-to-peer network with an advanced two-tier routing algorithm. The first routing algorithm is probabilistic, using a modification of Bloom filters. Each site has a filter expressing the data stored on that site. Sites also maintain filters for each outgoing connection, and the probabilistic algorithm routes queries from site to site following these filters. The second algorithm is a deterministic algorithm that is used if the probabilistic algorithm fails. Deterministic routing is based on hierarchical routing as suggested by Plaxton et al. [80]. This algorithm uses hashing on the GUID to create a global routing tree. To avoid problems with churn, the tree is replicated using different salts for the hashing function. The unstructured system allows data objects to be stored on any site.

Bigtable

Google's Bigtable [22] is another large-scale distributed data storage system. The data model of Bigtable is close to relational databases, but tuples are not stored or accessed as one unit. Instead, a row and column key is used for both read and write operations. Bigtable also supports a grouping of columns into what is called column families. Column families are part of the table schema, but within each family new columns can be created just by inserting data into the cell given by row, column family and column name. Access control is done on the column family level.

The query model is simple. A client is allowed to request a certain cell in the table, given by row and a column key consisting of column family and column name. The table can also be scanned, filtering on column family.

Bigtable uses a central server to provide locking services. This server has a number of passive replicas ready to take over in case of a server crash. Each table is stored as a three-level tree where the leafs are table fragments and the internal nodes of the tree are pointers to these leaves. The root node of the tree has pointers to these internal nodes. The central server has the global catalog with information about where the root for each table is located.

Apache HBase [47] is an open-source Bigtable implementation based on Hadoop.

Cloudy

Many cloud data storage systems have appeared that have different data and consistency models. Cloudy [58] is a storage system that tries to give configurable consistency and availability guarantees. Cloudy has a modular architecture and provides a number of storage engines for both main memory and disk storage.

On top of the storage layer are a number of internal modules for message routing, data partitioning, load balancing and site membership services. The external interface is also modular, providing modules for SQL, XQuery and key-value store access.

Cloudy provides a site membership service called Cloudburst. The Cloudburst service allows Cloudy to add and remove sites as the load changes, allowing the system to scale automatically.

ES^2

The Elastic Data Storage System ES^2 [19] is another storage system for the cloud. While most cloud data storage systems are key-value stores, ES^2 has a relational data model. The relations are partitioned both horizontally and vertically and stored in Hadoop's HDFS. Indices and metadata are also stored in HDFS.

The query processor supports both online transaction processing (OLTP) and online analytical processing (OLAP) queries. For OLTP queries, ES^2 provides get, put and delete functions. The get and delete functions work on sets of tuples so that one can query a table for multiple attributes and multiple tuples in one operation. This is different from the access model of Bigtable, which only provides single cell read and write operations.

The OLAP query processor provides scan methods, but unlike MapReduce, it provides index scans so that it is not necessary to scan the whole table. To enable OLTP and OLAP operations to run simultaneously, ES^2 stores multiple versions of tuples. OLTP operations access the latest version, while OLAP queries use older versions.

CouchDB

Apache CouchDB [29] is a document-oriented storage system. Semistructured documents are the primary data unit. These documents consist of a number of fields, attachments and metadata. Documents are of varying size and format and identified using a document ID.

Lookups can be done using the document ID or using views, which are a method to filter and aggregate documents. Views are defined using JavaScript functions that take documents as input and produce any number of rows in a view for each document. CouchDB optimization is largely concentrated around efficient view management, including maintaining indices of documents belonging to views.

Multi-version concurrency control is used to allow different applications to read and write the same document simultaneously, and the database is always kept in a consistent state, so that only a minimum of recovery is necessary after a crash.

Dynamo

Amazon's Dynamo [33] is a key-value store for cloud services with high availability demands. The interface is simple and consists of only a get and a put method that accesses data based on the primary key. No operations span more than one key-value pair, and it is not possible to combine multiple get or put request into transactions.

Data consists of a key and value, each an array of bytes that is not interpreted by Dynamo. Data items are stored in a circular data space, similar to that of Chord [92]. Each site is assigned responsibility for data items with keys hashing to an address between the site's address and the address of the site's predecessor in the ring. In order to balance the load between sites, each site is assigned multiple addresses, and therefore responsibility for multiple address ranges. Replication is done by storing copies on successor sites.

Voldemort

Voldemort [99] is another key-value store. Like Dynamo, it has a simple interface providing only primary key access, uses hashing to decide which site stores data items, and uses successors in the circular data space to replicate data.

The data model is more complex than that of Dynamo. In addition to uninterpreted byte arrays, Voldemort has data types for strings of text and serialized objects, including Java and JavaScript objects.

2.5.2 Query Systems

The query systems provide query capability, but not permanent storage. Query processing is done on data that is non-permanently stored by the system or produced on demand. Sensor networks fall into the last category.

PIER

The Peer-to-Peer Information Exchange and Retrieval system (PIER) [50, 51] is a general-purpose query network for relational data. PIER uses a DHT as overlay network for storing tuples. These tuples will time out after a set time, and sites that want to keep data in the PIER network has to republish their tuples regularly.

The interface to the overlay network is algorithm agnostic, and can be used with different DHT algorithms. PIER has been tested with several algorithms, such as CAN [83], Chord [92] and Bamboo [86].

PIER objects are identified by an identifier consisting of a namespace, a key and a distinguishing suffix. The namespace is the name of the table, or in some cases the name of a temporary result. The key is the table's key attributes. These two parts are combined and used as input to the DHT's hashing function. An additional suffix is used to make the identifiers unique in case two tuples hash to the same DHT key. Tuples are stored as serialized Java objects and can use the full Java type system for storing data.

Since tuples are not permanently stored in the network, they have to be refreshed regularly. A time-to-live value is associated with each tuple, and sites have to renew tuples before they time out. If a site disappears from the network, its tuples eventually time out and are removed. To avoid this, it must hand its data over to another site before leaving the network.

A special dataflow language called UFL is used to specify queries as relational algebra graphs. The client is considered to be outside the PIER network and contacts a site in the network to pose its query. The site that is contacted is called the proxy site and acts as coordinator for the query. The UFL query is transformed into a set of Java objects representing the algebra operations. The coordinator site distributes this graph of Java objects to the other sites. When a site receives a query on this form, it starts processing it and producing answer tuples. The tuples are continuously forwarded to the coordinator, until the query stops after a time limit specified in the query.

Queries are distributed using a distribution tree. There exists one such tree in the PIER network, and this tree is used for all queries. Sites that connect to the network are added to this tree by sending a message to a hardcoded root hash key. This message is routed through the DHT according to the routing algorithm, and the first site that receives the message adds the site as its child in the distribution tree.

Astrolabe

Astrolabe [85] is a system for continuously running aggregation queries used for system monitoring. Astrolabe maintains aggregated data in a hierarchy of zones. Each zone stores and maintains aggregate values based on its sub-zones. At the lowest level, each site in the system constitutes a leaf.

The network structure in Astrolabe is unstructured. A site has a connection to its parent zone, and automatically becomes a member of all ancestor zones. Each site also has connections to a set of other sites in its own zone and may have connections

to other sites in the system.

Each site constitutes a leaf zone. In this zone, it has complete control over the data. In the higher tiers of the hierarchy, zones store and maintain aggregates over the data in sub-zones. The root of the hierarchy has aggregates that cover the whole system. A site querying the network will connect directly to the site that maintains the zone in question.

When a data item is updated in a leaf zone, an aggregate value in the leaf zone may change. Updates are distributed to other parts of the network using a gossip protocol that guarantees eventual consistency. Using this protocol, a site randomly selects another site with which it exchanges information about its closest ancestor zone. Each aggregate value is marked with a time stamp so that the gossip protocol knows which value is the newest. After this exchange, aggregates of ancestor zones are computed if necessary. This gives eventual consistency, but sites in the same zone may be inconsistent at any given time.

The gossip messages also include information about zone membership. This way, sites learn about new sites that join the system. This information is crucial to keeping the network stable as sites come and go.

Astrolabe processes queries formulated in SQL and provides ODBC and JDBC programming interfaces. Since Astrolabe only provides eventual consistency guarantees, a query may return old data and subsequent queries may return inconsistent results.

SDIMS

The Scalable Distributed Information Management System (SDIMS) [107] is an aggregation system similar to Astrolabe, but based on a DHT. Similar to Astrolabe, all sites are leaves in a hierarchy of groups. Each site stores a selected set of attributes and aggregation functions to calculate the aggregates of these attributes for higher levels of the hierarchy.

Where Astrolabe uses a gossip protocol to update sites, SDIMS passes the aggregates up and down the group hierarchy. The SDIMS API provides three functions: install, update and probe. These operations take a parameter describing the requested propagation in the hierarchy, from local to global. This allows data items that are frequently read and infrequently written to be updated globally, so that probes (queries) can be done locally. On the other hand, items that are much more frequently written to can be updated only locally, avoiding flooding the whole system when the value is most likely overwritten before the next time it is read.

TAG

The Tiny Aggregation Service (TAG) [63] is an aggregation service for wireless sensor networks. One site is appointed the root and broadcasts a message stating its own identifier and its level in the hierarchy. All sites that hear this and are not already in the hierarchy, adds the site as their parent and broadcast similar messages containing their own identifiers and levels, which is one more than the parent's level. As this process is repeated, the whole system forms a hierarchy. The messages are

repeated periodically to detect new sites and to reconnect sites that have not been able to contact their parent for a while.

The hierarchy is used to do aggregation in two phases. In the first phase, the query is propagated downwards through the hierarchy. Each leaf site then computes its reply and sends it to the parent site. An intermediate level site gathers replies from its children, combines it with its own result and ships the result up one level to its own parent. In the end, the reply reaches the root of the tree and is returned to the querying application.

The technique is not very tolerant to failures. If one site is unable to contact its parent, the whole subtree rooted at that site is lost from the reply. In [63], caching is proposed as a solution to increase the quality of the results. If a child has not replied within a given time limit, its parent site will use the cached value. Since a site is allowed to choose a new parent if it is unable to connect to its current parent, care must be taken so that the cache is invalidated before the child can appear in another subtree.

Cougar

The Cougar Project [34, 109, 110] is another aggregation system for sensor networks, and like TAG it also has a hierarchical structure where queries enter at the root site, but the approach to query processing is quite different.

At the top of the tree is the root site that issues all the queries. The internal nodes, called view nodes, of the hierarchy store pre-aggregated values for the next lower level, which is the leaf nodes with the actual sensors.

Queries are processed in a hybrid pull-push manner. View nodes proactively aggregate the values read from the leaves and queries are made against the view nodes. Two different types of connections are allowed: on-demand and proactive. In proactive links, information is pushed to update pre-aggregated values, while on-demand links require view nodes to pull the data from leaves.

Querying is a three phase process. In phase one, leaves push data to the view nodes, which compute the aggregate values. In phase two, the query is sent out from the root to the view nodes. In the third an final phase, the view nodes send their replies to the root. The root then merges the partial results of all the view nodes to get the final result.

2.5.3 Data Storage and Querying Systems

The combination of data storage and advanced querying is closer to the traditional DBMSs, but also these systems come in a number of variations.

Piazza

The Piazza system [44] is a peer-to-peer data management system for integrating data sources with heterogeneous schemas. Instead of requiring all sites to share a common schema, e.g., as PIER does, Piazza mediates between these schemas, allowing each site full autonomy over which data to store and schema to use.

Both unstructured and supernode overlay networks can be used, but sites are not free to connect to any other site. A connection between two sites implies a partial or full schema mapping between the sites. Creating such schema mappings are heavyweight operations, which means that new connections are not added frequently. Because of this, the system is assumed to have a low churn level with very few sites joining or leaving.

The focus of the Piazza project has been schema mediation and query reformulation according to schema mappings. Piazza sites store data in XML and issue queries in a language inspired by XQuery. Queries that are sent over the network are reformulated to reflect schema mappings as they propagate from one site to another. The result of a query is similarly rewritten to new schemas as it is passed back to the querying site.

Replication is possible in Piazza. A storage description defines what content a site should store, and this description can define parts of other sites' databases that should be replicated locally. Like network connections, updates to this description is not expected to be a very frequent operation, and hence not automated.

PeerDB

PeerDB [74, 75, 76] is a peer-to-peer relational DBMS using mobile agents in query processing. It is built on the BestPeer platform [73], which is a system for mobile agents in peer-to-peer networks. BestPeer uses a supernode network and PeerDB combines this with the MySQL DBMS for data storage.

Sites in a PeerDB system can have heterogeneous schemas. Each schema is described by keywords, and query processing uses standard information retrieval techniques to find probable candidates for matching tables. When a query is issued, it is first parsed to extract table and attribute names. These names are looked up in a local dictionary to find matches in the local database. Agents are also shipped off to neighboring sites to find matching tables and attributes there. Possible matches are returned to the querying site, where the user selects which tables are to be included in the final answer. After the user has made this choice, the agents on the matching sites rewrite the query to match the local schema and return tuples as reply to the query.

AmbientDB

AmbientDB [37] uses a DHT to provide a self-organizing peer-to-peer network of intelligent home appliances. These devices share information using a database system. The DHT serves both as a way to connect the devices and as an indexing mechanism for data stored in the database.

The challenges facing AmbientDB include mobile devices with limited networking bandwidth and computing resources. The devices may also disconnect frequently and stay disconnected for long periods of time. AmbientDB provides database services both for single devices that are currently away from the rest of the network and for the rest of the network. Devices that are able to connect to other devices

nearby can access each other's data, and synchronize by updating data items that have been updated while the devices where disconnected.

The self-organizing properties of AmbientDB makes it possible for devices to extend the database schema and data propagation strategies. E.g., a thermometer may extend the database schema to include temperature recordings that can be used by other devices to automatically adjust to the surroundings.

APPA

The Atlas Peer-to-Peer Architecture (APPA) [10] builds on a supernode or structured overlay network and provides a full stack of data storage and querying services. The APPA system consists of multiple layers. On the bottom layer, APPA provides a simple key-value store, and higher level layers build more advanced services. On the top level is services such as schema management, replication and query processing.

Schema mapping in APPA is different from the pairwise schema mappings in Piazza. In APPA, sites agree on a common global schema and express their tables as views of this schema. Querying is done against the local view, and this query is reformulated against the global schema before the set of sites with data relevant to the query are found.

Mariposa

Mariposa [94] is a distributed DBMS that uses economic models to solve optimization problems. Table fragments are considered a resource and are bought and sold during a bidding process that is used to decide which sites should participate in processing a given query.

A new query is given a budget to use on query processing. The sites bid for the execution of this query, and the economy is constructed in such a way that the more expensive query plans are more efficient. This means that a query that has been given a large budget is prioritized and can buy a more efficient execution. This bidding process includes the buying and selling of table fragments in order to move data closer to the processing sites.

ObjectGlobe

ObjectGlobe [17] is a distributed storage and query processing system where data, query operators and computing power is traded. Each site can offer a combination of data sets, query operators and computing power. A site that wants to execute a query combines components from several other sites and buys computing resources to process the query. A pipeline is constructed that ships intermediate results from one operator to another.

The difference between ObjectGlobe and Mariposa is that while Mariposa uses the economic model to improve querying performance, ObjectGlobe has a more direct connection to a real economy, where data sets, operator implementations and computing power are actually sold.

HadoopDB

HadoopDB [3] is a database middleware system for the cloud. It is built on top of Hadoop, an implementation of MapReduce [32]. Each site in the system has a local DBMS that manages storage, and these sites are connected by the Hadoop framework. The local database is integrated with the MapReduce framework so that it can be accessed similarly to Hadoop's own distributed file system, HDFS. A metadata catalog containing information about local DBMSs, data sets, partitioning and replication is stored in HDFS.

The Hadoop framework is used to coordinate tasks and to distribute query processing, and its failure resilience properties are used to achieve fault tolerance. A modified implementation of Hive [96] is used to transform a query from the SQL-like query language to MapReduce tasks. Query plans are constructed so that as much query processing as possible is done in the local DBMS on each site. By doing this, HadoopDB is able to maintain the failure resilience properties of Hadoop and much of the query optimization of the DBMS.

MongoDB

MongoDB [69] is document-oriented like CouchDB, but provides a much more advanced querying system. The basic data unit is a semistructured document, and documents are grouped into collections. Typically, there is a collection for each document type.

Data can be fragmented and replicated to increase availability and performance. MongoDB also provides automatic load balancing for query load and data distribution. Queries are posed in an imperative language by defining filters for a scan over a document collection, and B-tree indices are used to increase performance.

VoltDB

A more traditional relational DBMSs is provided by VoltDB [100]. VoltDB is a distributed main memory DBMS that uses a traditional relational data model with schemas defined in SQL. However, it is not as traditional when it comes to querying. Each site runs a single-threaded server process. Because of single-threading, no locking is used and no concurrency issues occur. All access to the tables are via stored procedures that constitute a microtransaction.

Being a main memory DBMS, replication is used to provide data persistence. Active replication to other sites in the same data center is used to protect against single site failures. Passive replication to other data centers is used to protect against larger outages.

Chapter 3

Contributions

This thesis is a collection of published papers describing work on query processing in distributed DBMSs. This chapter explains how the individual papers fit together and provides details about each author's contributions.

3.1 Research Topics

The research can be split up into four parts. First, a study of the effects of failures on aggregation queries, described in Section 3.1.1. Section 3.1.2 describes the research on how data is distributed among sites in the system and how they are looked up when needed. Section 3.1.3 describes the research done on caching to improve query execution. Finally, Section 3.1.4 describes the DASCOSA-DB distributed DBMS prototype that was built partly as a result of the work conducted for this thesis.

3.1.1 The Effect of Failures on Aggregation Queries

In large distributed systems, it is expected that sites fail during query processing. If the failure rate of individual sites is kept constant, the failures will become more and more frequent as new sites are added to the system. This means that large distributed systems must be planned with failures in mind. In a database management system, these failures will affect ongoing queries, and if they are not detected, they may alter the result of these queries.

In Paper A, the effect of failures in a peer-to-peer system on aggregation queries is studied. A reduction tree is created in a peer-to-peer network and queried for typical aggregate values: minimum, maximum, sum, count and average.

The experiments show that the aggregate functions behave differently with regard to failures. Some aggregation functions are more affected than others. Some aggregation functions depend on all tuples to produce their result, while others only require the one most important tuple. This leads to different methods for handling failures for the different types of aggregation functions.

The main finding of the paper is that replication is not always the best answer to the challenges introduced by failures and that other design changes, such as

adjusting the branching factor of reduction trees, may be more efficient measures against data loss.

3.1.2 Data Placement and Query Localization

Given a query in a distributed DBMS, the system must be able to efficiently locate the data that is needed to answer the query, i.e., it must split the sites into the set of sites with relevant data and the set of sites without relevant data. This is necessary to avoid contacting every site on every query, which would swamp the system in messages very quickly.

Paper B proposes a distributed range index that stores ranges of index keys to reduce the amount of update messages for write heavy loads. Index updates are only issued for keys that do not fall within one of the existing ranges in the index. This reduces the number of update messages greatly, without affecting the use of the index for query localization. In addition, the proposed indexing method allows range lookups, which is essential for query localization.

While localization is essential to query processing, data placement strategies are optional. However, the placement strategies are important when considering the performance of query processing. It takes longer time to access data over a network connection than accessing it locally, but accessing it locally usually means that it has to be transferred before operations start, which is also a cost that should be considered. For some queries it might be better to use remote accesses, while other queries may do better with first moving the data so that they can be accesses locally.

Another aspect of data placement is the use of replication. With multiple replicas, multiple sites can benefit from local read access. However, the price to pay is more costly updates, since all replicas must be updated.

Paper D proposes a dynamic fragmentation and replication method that moves data to sites that use them if it is deemed beneficial to the system as a whole. Replicas are also created, migrated and deleted if it benefits the currently running queries. This method looks at the global query load as a whole, but locally on a single table fragment at a time. This means that it improves performance of the global query load without much coordination effort.

3.1.3 Caching of Query Results

Caching is used in almost all types of systems to reduce processing time, including DBMSs. In a centralized system, disk blocks are cached to provide quick access to frequently used data, and the same is possible in data shipping distributed systems. However, in a query shipping database management system, there are no fixed-size blocks transported between sites. Rather, the tuples passed between sites are the result of a sequence of relational algebra operators. Caching of these are not as straight forward as caching of disk blocks.

Paper E describes how semantic caching can be implemented in a distributed query shipping system. Semantic caching tags a cached item with semantic information that describes it. This information can later be used to match this cache

entry to new queries. The caching method allows later queries to use the whole cache entry as is, to use only a subset of it, or to extend it using a remainder query. With these options, a query can save a significant amount of processing time.

A special type of semantic caching is studied in Paper F. The top-k query operator allows for significant new options in using cached results. The result of a top-k query over a set of attributes of a table can be used to answer any other top-k query over the same attributes. This paper shows that significant savings can be achieved if the results of previous queries are cached.

3.1.4 DASCOSA-DB

Much of the research included in this thesis has used the DASCOSA-DB distributed DBMS prototype as a framework for implementing research ideas. DASCOSA-DB is a system that increases scalability by reducing coupling between its sites. Each site is to a large degree autonomous and makes its own decisions about storage and query processing. By making sites autonomous, there is less need for coordination, and it is easier to scale the system.

With increasing system size comes an increased failure rate. DASCOSA-DB is built to withstand site and network failures by relying on peer-to-peer technology. Papers C and G describes the details of DASCOSA-DB.

Since the DASCOSA-DB prototype became available, it has been used as a basis for the experiments conducted for Papers D–F. New methods have been implemented as extensions to DASCOSA-DB and experiments have been conducted on this implementation.

3.2 Published Papers

This section contains a list of all papers in Part II with publication details, abstracts and statements of each author's contributions to the paper.

3.2.1 Paper A: Robust Aggregation in Peer-to-Peer Database Systems

This paper was published in the proceedings of the 12th International Database Engineering & Applications Symposium (IDEAS). The paper was presented at IDEAS 2008 in Coimbra, Portugal, September 10–12, 2008.

Abstract

Peer-to-peer database systems (P2PDBs) aim at providing database services with node autonomy, high availability and loose coupling between participating nodes by building the DBMS on top of a peer-to-peer network. A key feature of current peer-to-peer systems is resilience to churn in the overlay network layer. A major challenge in P2PDBs is to provide similar robustness in the data and query processing layer. In this paper we in particular describe how aggregation queries in P2PDBs can be

handled in order to reduce the impact of churn on accuracy of results. We perform a formal study of data loss and accuracy of such queries, and describe new approaches that increase the accuracy of aggregation queries in P2PDBs under churn.

Statement of Contributions

Norvald H. Ryeng was the main contributor of the work described in this paper. Kjetil Nørvåg had the role of an active contributor/supervisor, including discussing the ideas and editing the paper.

3.2.2 Paper B: RIPPNET: Efficient Range Indexing in Peer-to-Peer Networks

This paper was published in the proceedings of the 3rd IEEE International Conference on Digital Information Management (ICDIM). The paper was presented at ICDIM 2008 in London, UK, November 13–16, 2008.

Abstract

Write-heavy applications present a challenge to peer-to-peer indexing methods which need to update the index for each write operation. The costs incurred when the distributed index is updated becomes a bottleneck. Current distributed indexing methods are designed for indexing and retrieving single tuples, giving a very high update cost. In this paper we present a new approach to efficient peer-to-peer range indexing that employs indexing of ranges to reduce average update costs as well as providing efficient data localization and decoupling from data placement policies. Based on results from experiments, we demonstrate the applicability and significantly reduced update cost of the new approach.

Statement of Contributions

Norvald H. Ryeng was the main contributor of the work described in this paper. Kjetil Nørvåg had the role of an active contributor/supervisor, including discussing the ideas and editing the paper.

3.2.3 Paper C: Efficient and Robust Database Support for Data-Intensive Applications in Dynamic Environments

This paper was published in the proceedings of the 25th International Conference on Data Engineering (ICDE). The paper was presented and the system demonstrated at ICDE 2009 in Shanghai, China, March 29–April 2, 2009.

Abstract

Requirements from new types of applications call for new database system solutions. Computational science applications performing distributed computations on Grid

networks with requirements for efficient storage and query solutions are now emerging. For this purpose we have developed DASCOSA-DB, a P2P-based distributed database system, which in addition to providing location-transparent storage and querying, also includes novel features like efficient partial restart of queries and redistribution of query operators in the context of failure, dynamic refragmentation and allocation, and distributed semantic caching. In this demo, the novel features will be demonstrated, combined with a more general description of the architecture and demonstration of the distributed query processing capabilities.

Statement of Contributions

Jon Olav Hauglid had the largest contribution to DASCOSA-DB. He designed and implemented most of the system. Norvald H. Ryeng contributed the semantic caching method and its implementation and also contributed to the dynamic fragment management and the overall design of DASCOSA-DB. Kjetil Nørvåg had the role of an active contributor/supervisor, including many discussions and editing the paper.

3.2.4 Paper D: DYFRAM: Dynamic Fragmentation and Replica Management in Distributed Database Systems

This paper was published in Distributed and Parallel Databases 28(2–3), pages 157–185, 2010.

Abstract

In distributed database systems, tables are frequently fragmented and replicated over a number of sites in order to reduce network communication costs. How to fragment, when to replicate and how to allocate the fragments to the sites are challenging problems that has previously been solved either by static fragmentation, replication and allocation, or based on a priori query analysis. Many emerging applications of distributed database systems generate very dynamic workloads with frequent changes in access patterns from different sites. In such contexts, continuous refragmentation and reallocation can significantly improve performance. In this paper we present DYFRAM, a decentralized approach for dynamic table fragmentation and allocation in distributed database systems based on observation of the access patterns of sites to tables. The approach performs fragmentation, replication, and reallocation based on recent access history, aiming at maximizing the number of local accesses compared to accesses from remote sites. We show through simulations and experiments on the DASCOSA distributed database system that the approach significantly reduces communication costs for typical access patterns, thus demonstrating the feasibility of our approach.

Statement of Contributions

Jon Olav Hauglid was the main contributor to the work described in this paper. The original idea was further developed in discussions with Norvald H. Ryeng, who also participated in writing the paper. The implementation in DASCOSA-DB was done by Jon Olav Hauglid in tandem with Norvald H. Ryeng testing and debugging it and performing the experiments in DASCOSA-DB. Kjetil Nørvåg had the role of an active contributor/supervisor, including discussing the ideas and editing the paper.

3.2.5 Paper E: Site-Autonomous Distributed Semantic Caching

This paper was published in the proceedings of the 26th Symposium on Applied Computing (SAC). The paper was presented at SAC 2011 in Taichung, Taiwan, March 21–24, 2011.

Abstract

Semantic caching augments cached data with a semantic description of the data. These semantic descriptions can be used to improve execution time for similar queries by retrieving some data from cache and issuing a remainder query for the rest. This is an improvement over traditional page caching, since caches are no longer limited to only base tables but are extended to contain intermediate results. In large-scale distributed database systems, using a central server with complete knowledge of the system will be a serious bottleneck and single point of failure. In this paper, we propose a distributed semantic caching method where sites make autonomous caching decisions based on locally available information, thereby reducing the need for centralized control. We implement the method in the DASCOSA-DB distributed database system prototype and use this implementation to do experiments that show the applicability and efficiency of our approach. Our evaluation shows that execution times for queries with similar subqueries are significantly reduced and that overhead caused by cache management is marginal.

Statement of Contributions

Norvald H. Ryeng was the main contributor to the work described in this paper. Jon Olav Hauglid advised on the implementation in DASCOSA-DB and contributed to the discussions. Kjetil Nørvåg had the role of an active contributor/supervisor, including discussing the ideas and editing the paper.

3.2.6 Paper F: Efficient Distributed Top-*k* Query Processing with Caching

This paper was published in the proceedings of the 16th Conference on Database Systems for Advanced Applications (DASFAA). The paper was presented at DAS-

FAA 2011 in Hong Kong, China, April 22–25, 2011.

Abstract

Recently, there has been an increased interest in incorporating in database management systems rank-aware query operators, such as top-k queries, that allow users to retrieve only the most interesting data objects. In this paper, we propose a cache-based approach for efficiently supporting top-k queries in distributed database management systems. In large distributed systems, the query performance depends mainly on the network cost, measured as the number of tuples transmitted over the network. Ideally, only the k tuples that belong to the query result set should be transmitted. Nevertheless, a server cannot decide based only on its local data which tuples belong to the result set. Therefore, in this paper, we use caching of previous results to reduce the number of tuples that must be fetched over the network. To this end, our approach always delivers as many tuples as possible from cache and constructs a remainder query to fetch the remaining tuples. This is different from the existing distributed approaches that need to re-execute the entire top-k query when the cached entries are not sufficient to provide the result set. We demonstrate the feasibility and efficiency of our approach through implementation in a distributed database management system.

Statement of Contributions

Norvald H. Ryeng was the main contributor to the work described in this paper. Akrivi Vlachou and Christos Doulkeridis had the role of active contributors, including discussing the ideas and editing the paper. Kjetil Nørvåg had the role of an active supervisor.

3.2.7 Paper G: The DASCOSA-DB Grid Database System

This paper was published as a chapter in the book Grid and Cloud Database Management, Giovanni Aloisio and Sandro Fiore (editors), Springer-Verlag, 2011.

Abstract

Computational science applications performing distributed computations using grid networks are now emerging. These applications have new and demanding requirements for efficient query processing. In order to meet these requirements, we have developed the DASCOSA-DB distributed database system. In this chapter, a detailed overview of the architecture and implementation of DASCOSA-DB is given, as well as a description of novel features developed in order to better support typical data-intensive applications running on a grid system: fault-tolerant query processing, dynamic refragmentation, allocation and replication of data fragments, and distributed semantic caching.

Statement of Contributions

Jon Olav Hauglid had the largest contribution to DASCOSA-DB. He designed and implemented most of the system. Norvald H. Ryeng contributed the semantic caching method and its implementation and also contributed to the dynamic fragment management and the overall design of DASCOSA-DB. Kjetil Nørvåg had the role of an active contributor/supervisor, including many discussions and editing the paper.

Chapter 4

Concluding Remarks

This thesis has examined some selected techniques for improving execution time and cost of queries in large distributed DBMSs. The improvements are made in the areas of data placement and localization and caching of results and intermediate results of queries.

The study of aggregation queries presented in this thesis shows that different queries are affected differently by churn and explains why replication is not necessarily the best method to improve churn resilience in aggregation processing. Reduction tree branching factor and selective replication of parts of the tree are identified as ways to improve accuracy without increasing query costs too much.

Data placement has a significant role in query processing. The work presented in this thesis has shown how data can be fragmented, allocated and replicated to adapt to a constantly changing workload and presents results that show a large reduction in the number of remote data accesses and a corresponding reduction in query execution time.

Peer-to-peer networks put restrictions on data lookup, and this thesis has presented a range indexing method that decouples data placement and indexing in order to provide support for range and cover queries. The method reduces the cost of inserts and updates since index records do not have to be updated after every such operation.

A method for semantic caching of intermediate results in a query shipping distributed DBMS has been proposed, along with experimental results that show how execution time decreases when these caches are enabled. A new caching mechanism for top-k building on the semantic cache has also been presented. The experimental evaluation shows a significant reduction in the number of remote accesses.

The methods presented in this thesis are a step on the way to make relational distributed DBMSs for grid and cloud computing scale by making sites more autonomous and thereby reducing coordination overhead. To match the elasticity of the grids and clouds themselves, the methods adapt automatically to changing workloads.

4.1 Evaluation of Contributions

The main research question for this thesis was defined in Section 1.2.1 as

> *How can query processing in large distributed database management systems be made more efficient?*

This question was then narrowed down to four more specific research questions. This section evaluates the contributions based on these four questions.

4.1.1 How Are Aggregation Operations Influenced by Site Failures?

This question is addressed in Paper A. Aggregation functions are affected differently, which calls for different measures to increase resilience to site and network failures. Other papers suggest using replication of the aggregation process, but the results show that some aggregation functions respond better to adjusting the branching factor of reduction trees. In cases where replication is beneficial, it is not always necessary to replicate the whole tree, but only the most important nodes.

4.1.2 How Can Data Placement Adapt to Constantly Changing Workloads?

This question is addressed in Papers C, D and G. By keeping statistics of remote and local accesses to regions of a table fragment, tables are refragmented, reallocated, and replicated in order to provide as many local reads as possible, taking the cost of transferring fragments into account. The result is a strategy for dynamically making data placement decisions in a way that reduces the execution time for queries and is able to adapt to changing workloads.

4.1.3 How Can Data Be Located If Sites Are Given Autonomy over Storage Decisions?

This question is addressed in Paper B. Instead of tuple indexing, a range index is used to identify relevant data fragments and sites that contain data relevant to a query. By decoupling the index from data placement, sites are allowed full autonomy over storage decisions, while retaining the ability to efficiently locate data.

4.1.4 How Can Caching of Intermediate Results Be Used to Speed up Query Processing?

This question is addressed in Papers C and E–G. The semantic caching allows intermediate and final results to be cached and reused for similar queries. Both the general technique of Paper E and the adapted implementation for top-k queries in Paper F show reduced execution times.

4.2 Future Work

The main research question is wide, and the work for this thesis had to be narrowed down to only a few specific methods. This leaves a number of research opportunities unvisited.

As Paper A shows, there are opportunities for operator specific improvements. Query operators differ, and the methods for improving failure resilience affect operators differently. More efficient and less costly methods may be chosen by looking into the specifics of each operator instead of more general approaches. Paper B provides a distributed range indexing method that can be extended to answer some aggregation queries. Such an extension could also be used to provide approximations of data stored on failed sites, which could increase accuracy further.

The data placement method proposed in Paper D is adaptive, but still relies on a few manually set parameters. Adaptive adjustment of these parameters is beneficial since that would offer a completely automatic and self tuning data placement algorithm.

The caching method described in Paper E may benefit from a query planner that invests in the cache by creating suboptimal query plans to increase future cache hit rates. The specific caching method of Paper F still has to be fully integrated with the general caching method of Paper E by resolving cache replacement policy incompatibility between the specific and the general caching method. There is also potential for further improvements to the individual cache replacement policies.

Bibliography

[1] M. Abdallah and H. C. Le. Scalable range query processing for large-scale distributed database applications. In *Proceedings of PDCS*, 2005.

[2] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A self-organizing structured P2P system. *SIGMOD Record*, 32(3):29–33, 2003.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2:922–933, August 2009.

[4] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), 2000.

[5] D. Agrawal, A. E. Abbadi, F. Emekçi, and A. Metwally. Database management as a service: Challenges and opportunities. In *Proceedings of ICDE*, 2009.

[6] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont report on database research. *SIGMOD Record*, 37:9–19, September 2008.

[7] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: Workload as a sequence. In *Proceedings of SIGMOD*, 2006.

[8] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, 2004.

[9] I. Ahmad, K. Karlapalem, Y.-K. Kwok, and S.-K. So. Evolutionary algorithms for allocating data in distributed database systems. *Distributed and Parallel Databases*, 11(1):5–32, 2002.

[10] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Design and implementation of Atlas P2P architecture. In *Global Data Management*, 2006.

[11] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of VLDB*, 2003.

[12] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13(3):263–304, 1988.

[13] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of SODA*, 2003.

[14] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM*, 2004.

[15] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of SoCC*, 2010.

[16] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering Bulletin*, 27(2):11–18, 2004.

[17] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[18] A. Brunstrom, S. T. Leutenegger, and R. Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Proceedings of CIKM*, 1995.

[19] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. ES^2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of ICDE*, 2011.

[20] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, December 2010.

[21] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of VLDB*, 1986.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of OSDI*, 2006.

[23] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE*, 1995.

[24] B. Chidlovskii, C. Roncancio, and M.-L. Schneider. Semantic cache mechanism for heterogeneous web querying. *Computer Networks*, 31(11–16):1347–1360, 1999.

[25] B. Ciciani, D. Dias, and P. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):247–261, June 1990.

[26] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[27] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of SIGMOD*, 1988.

[28] A. L. Corcoran and J. Hale. A genetic algorithm for fragment allocation in a distributed database system. In *Proceedings of SAC*, 1994.

[29] Apache CouchDB: Technical overview. `http://couchdb.apache.org/docs/overview.html`.

[30] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of WebDB*, 2004.

[31] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, 1996.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[33] 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. *SIGOPS Operating Systems Review*, 41:205–220, October 2007.

[34] A. Demers, J. Gehrke, R. Rajaraman, N. Trigoni, and Y. Yao. The Cougar project: A work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.

[35] T. Didriksen, C. A. Galindo-Legaria, and E. Dahle. Database de-centralization — a practical approach. In *Proceedings of VLDB*, 1995.

[36] C. Doulkeridis, A. Vlachou, K. Nørvåg, Y. Kotidis, and M. Vazirgiannis. Multidimensional routing indices for efficient distributed query processing. In *Proceeding of CIKM*, 2009.

[37] W. Fontijn and P. Boncz. AmbientDB: P2P data management middleware for ambient intelligence. In *Proceedings of PERCOMW*, 2004.

[38] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of DOLAP*, 2004.

[39] J. Gao and P. Steenkiste. Efficient support for range queries in DHT-based systems. Technical Report CMU-CS-03-215, Carnegie Mellon University, 2003.

[40] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The Pig experience. *Proceedings of the VLDB Endowment*, 2:1414–1425, August 2009.

[41] B. Gavish and O. R. L. Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33(2):177–189, 1990.

[42] The Gnutella homepage. `http://www.gnutella.com/`.

[43] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of CIDR*, 2003.

[44] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[45] T. Hara and S. K. Madria. Data replication for improving data accessibility in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(11):1515–1532, 2006.

[46] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of USITS*, 2003.

[47] The Apache HBase homepage. `http://hbase.apache.org/`.

[48] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of SPAA*, 2002.

[49] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of VLDB*, 1990.

[50] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: An Internet-scale query processor. In *Proceedings of CIDR*, 2005.

[51] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, 2003.

[52] Y. Ioannidis. Query optimization. In *The Computer Science and Engineering Handbook*, pages 1038–1054. CRC Press, 1996.

[53] M. Ivanova, M. L. Kersten, and N. Nes. Adaptive segmentation for scientific databases. In *Proceedings of ICDE*, 2008.

[54] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of VLDB*, 2005.

[55] B. T. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3):302–331, 2006.

[56] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[57] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag. Cache investment: Integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000.

[58] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: A modular cloud storage system. *Proceedings of the VLDB Endowment*, 3(2):1533–1536, 2010.

[59] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*, 2000.

[60] T. Kyte. On Oracle database 11g. *Oracle Magazine*, XXI(5), 2007.

[61] D. Lee and W. W. Chu. Semantic caching via query matching for web sources. In *Proceedings of CIKM*, 1999.

[62] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of SIGMOD*, 2002.

[63] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[64] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS*, 2002.

[65] S. Menon. Allocating fragments in distributed databases. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):577–585, 2005.

[66] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Record*, 30(2):307–318, 2001.

[67] A. Mondal, S. K. Madria, and M. Kitsuregawa. CADRE: A collaborative replica allocation and deallocation approach for mobile-P2P networks. In *Proceedings of IDEAS*, 2006.

[68] A. Mondal, K. Yadav, and S. K. Madria. EcoBroker: An economic incentive-based brokerage model for efficiently handling multiple-item queries to improve data availability via replication in mobile-P2P networks. In *Proceedings of DNIS*, 2010.

[69] The MongoDB homepage. http://www.mongodb.org/.

[70] MySQL 5.5 reference manual, Chapter 7.9.3: The MySQL query cache, 2010.

[71] The Napster homepage. http://www.napster.com/.

[72] Napster protocol specification. http://opennap.sourceforge.net/napster.txt, April 2000.

[73] W. S. Ng, B. C. Ooi, and K.-L. Tan. BestPeer: A self-configurable peer-to-peer system. In *Proceedings of ICDE*, 2002.

[74] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of ICDE*, 2003.

[75] B. C. Ooi, Y. Shu, and K.-L. Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 32(3):59–64, 2003.

[76] B. C. Ooi, K.-L. Tan, A. Zhou, C. H. Goh, Y. Li, C. Y. Liau, B. Ling, W. S. Ng, Y. Shu, X. Wang, and M. Zhang. PeerDB: Peering into personal databases. In *Proceedings of SIGMOD*, 2003.

[77] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 2nd edition, 1999.

[78] P. Padmanabhan, L. Gruenwald, A. Vallur, and M. Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *VLDB Journal*, 17(5):1143–1164, 2008.

[79] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Proceedings of EDBT*, 2006.

[80] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of SPAA*, 1997.

[81] A. D. Popescu, D. Dash, V. Kantere, and A. Ailamaki. Adaptive query execution for data management in the cloud. In *Proceedings of CloudDB*, 2010.

[82] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *Proceedings of SIGMOD*, 2002.

[83] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*, 2001.

[84] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of MobiCom*, 2000.

[85] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Compututer Systems*, 21(2):164–206, 2003.

[86] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of USENIX ATC*, 2004.

[87] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware*, 2001.

[88] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems*, 10(1):29–56, 1985.

[89] D.-G. Shin and K. B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Transactions on Software Engineering*, 17(9):872–883, 1991.

[90] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Proceedings of P2P*, 2005.

[91] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proceedings of ICDE*, 1996.

[92] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of SIGCOMM*, 2001.

[93] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: Friends or foes? *Communications of the ACM*, 53:64–71, January 2010.

[94] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[95] A. Tamhankar and S. Ram. Database fragmentation and allocation: An integrated methodology and case study. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 28(3):288–305, 1998.

[96] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2:1626–1629, August 2009.

[97] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *Proceedings of ICDE*, 2011.

[98] T. Ulus and M. Uysal. Heuristic approach to dynamic data allocation in distributed database systems. *Pakistan Journal of Information and Technology*, 2(3):231–239, 2003.

[99] The Voldemort homepage. `http://www.project-voldemort.com/`.

[100] The VoltDB homepage. http://voltdb.com/.

[101] G. Weikum, C. Hasse, A. Moenkeberg, and P. Zabback. The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5):381–432, 1994.

[102] Wikipedia: FastTrack. http://en.wikipedia.org/wiki/FastTrack.

[103] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proceedings of PODS*, 1992.

[104] E. Wong and R. H. Katz. Distributing a database for parallelism. *SIGMOD Record*, 13(4):23–29, 1983.

[105] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and parallel DBMS. In *Proceedings of SIGMOD*, 2010.

[106] P. Yalagandula and J. C. Browne. Solving range queries in a distributed system. Technical Report TR-04-18, Department of Computer Sciences, University of Texas at Austin, 2004.

[107] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of SIGCOMM*, 2004.

[108] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of SIGMOD*, 2009.

[109] Y. Yao and J. Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[110] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proceedings of CIDR*, 2003.

[111] C. Zhang, A. Krishnamurthy, and R. Y. Wang. SkipIndex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton University Computer Science Department, 2004.

[112] M. Zhang and K.-L. Tan. Supporting rich queries in DHT-based peer-to-peer systems. In *Proceedings of WETICE*, 2003.

[113] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *Proceedings of IPTPS*, 2006.

[114] M. Zhou, R. Zhang, W. Qian, and A. Zhou. GChord: Indexing for multi-attribute query in P2P system with low maintenance cost. In *Proceedings of DASFAA*, 2007.

[115] D. C. Zilio et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of VLDB*, 2004.

Part II

Published Papers

Paper A

Robust Aggregation in Peer-to-Peer Database Systems

Norvald H. Ryeng and Kjetil Nørvåg.
In *Proceedings of IDEAS*, 2008.

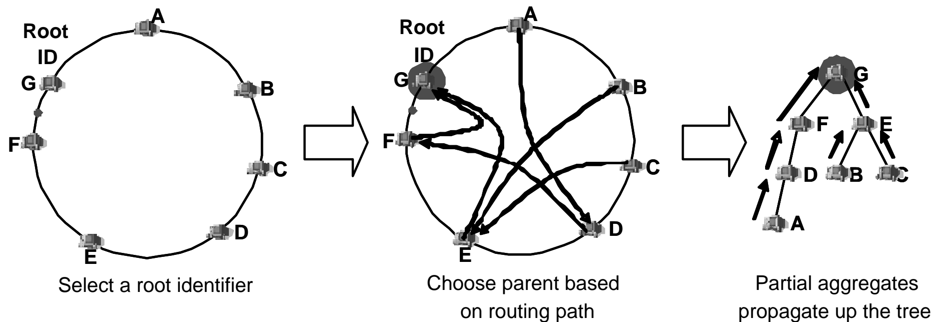


Figure A.1: Hierarchical aggregation using a DHT.

Abstract

Peer-to-peer database systems (P2PDBs) aim at providing database services with node autonomy, high availability and loose coupling between participating nodes by building the DBMS on top of a peer-to-peer network. A key feature of current peer-to-peer systems is resilience to churn in the overlay network layer. A major challenge in P2PDBs is to provide similar robustness in the data and query processing layer. In this paper we in particular describe how aggregation queries in P2PDBs can be handled in order to reduce the impact of churn on accuracy of results. We perform a formal study of data loss and accuracy of such queries, and describe new approaches that increase the accuracy of aggregation queries in P2PDBs under churn.

A.1 Introduction

A key feature of current peer-to-peer routing mechanisms is resilience to *churn*, the effect of nodes constantly joining and parting from the network. Nodes leaving the network, either because of a planned shutdown or because of a node or network failure, will generally not interfere with the message passing capability of the network; network traffic is routed through other nodes until a disconnected node reconnects.

A major challenge in peer-to-peer database systems (P2PDB) is to provide similar robustness in the data and query processing layer. When a node is disconnected, the data stored at that node is also inaccessible. In some cases it may be possible for nodes to hand over their data before disconnecting, but in case of node and network failures, data may become inaccessible without warning. The failure rate of a large distributed system is such that the system cannot expect all nodes to be accessible at all times, so waiting for disconnected nodes to reconnect is not generally an option. Instead, when nodes fail, query processing has to be based on partial data.

The typical method for doing aggregation in P2PDBs is to use a reduction tree, as illustrated in Fig. A.1. In this way, the nodes at the leaves of the tree start aggregating over their local database, and each leaf node sends its partial aggregates to its parent node. An intermediate-level node gathers partial results from its chil-

dren and merges these results with the result from the local database. The result of the merge is sent upwards in the tree to the parent node. This passing of partial aggregates continues all the way to the root node, which merges and evaluates the partial aggregates to get the final result of the query.

The problem with this approach is that failure of internal nodes causes loss of data from all nodes below it in the hierarchy. Existing work propose to use replication of the aggregation process to counter this effect [5, 7]. In this paper we study in more detail the data loss in aggregation and show that costly replication is not necessarily the best way to improve the accuracy of results.

The analysis and techniques presented in this paper are applicable both for P2P systems based on distributed hash tables (e.g., Chord [15], CAN [10], Pastry [12]), as well as unstructured P2P-systems where tree-overlays can be created through flooding (e.g., Gnutella-like networks).

The main contributions of this paper are 1) a formal study of data loss in P2PDB aggregation queries, 2) new approaches that reduces the impact of churn on aggregation accuracy, and 3) an experimental study of the effect of various parameters in techniques used to reduce the impact of churn on aggregation accuracy.

The organization of the rest of the paper is as follows. In Section A.2 we give an overview of related work. In Section A.3 we perform a formal study of data loss and accuracy of query results in P2PDBs. In Section A.4 we discuss techniques for reducing data loss. In Section A.5 we present experimental results. Finally, in Section A.6, we conclude the paper and outline issues for further work.

A.2 Related Work

Although very popular for file-sharing applications and distributed computing, only a few P2PDBs and P2P data management systems have been realized so far. The most well-know systems include PIER, Piazza, APPA, and PeerDB:

- The Peer-to-Peer Information Exchange and Retrieval system (PIER) [5, 6] is a general-purpose query processor that executes relational queries in a DHT-based network. PIER does not implement persistent storage and relies on external data producers to insert and renew data.

- The Piazza system [4] is a peer data management system that mediates between heterogeneous schemas. Instead of requiring all nodes to share a common schema (as, e.g., PIER does), each node is allowed full autonomy in which data it wants to store and which schema to use.

- The APPA system [16], on the other hand, assumes that participating nodes will agree to a common schema description.

- PeerDB [9] also supports schema matching using agent technology to find relevant relations on other nodes.

Distributed aggregation queries are also performed in other variants of distributed computing systems and sensor network systems, for example:

- In Astrolabe [11], data is placed in a hierarchy of *zones*. Each zone stores and maintains aggregated data from its sub-zones, and at the lowest level, from *virtual leaf zones* constructed around single nodes.

- The Scalable Distributed Information Management System (SDIMS) [17] is an aggregation system similar to Astrolabe, but based on a DHT.

- The Tiny Aggregation Service (TAG) [8] is an aggregation service for wireless sensor networks. TAG is based on a hierarchical network which is also used as the reduction tree.

- The Cougar Project [3] also uses a hierarchical algorithm, but the approach is quite different from that of TAG. Queries are processed in a hybrid pull-push manner, where information is proactively updated at the second level view nodes, which then are queried by the root node.

In general, papers about aggregation queries in sensor networks are performed by best-effort algorithms where churn and accuracy are not taken into account. This contrasts to our work which makes adaption of parameters and choice of algorithm possible in order to achive high accuracy under churn. An exception to best-effort algorithms is the approach presented in [2], which is more robust to churn but at considerable cost for some aggregation operator types.

Related topics to churn-resistant aggregation are approaches to avoid/reduce the impact of cheaters/tampering and approximate query processing. For example, sensor networks are vulnerable to tampering, and networks could be attacked with the intent of giving erroneous answers to queries. In [13], algorithms are presented for aggregation despite compromised nodes.

When cost is more important than accuracy, approximate query processing can be employed. A sampling-based approach to aggregation query processing is proposed in [1].

In [14] the topic of accuracy in P2P aggregation was introduced.

A.3 Data Loss

In this section we investigate and formalize the notion of data loss in P2PDB aggregation queries using reduction trees. Formulas for evaluating the accuracy of query results are also presented.

A.3.1 Network Model

A peer-to-peer network $G = \langle V, E \rangle$ consists of a set V of nodes and a set E of network links between these nodes. Not all nodes and network links need to be fully functional at all times. Some nodes and network links may experience a failure, or may choose to part the network for a time. The total network is the network where V contains all nodes, both those currently in the network and those that are currently disconnected, and E contains all network links used by these nodes

to communicate with each other. The coming and going of nodes that occurs in peer-to-peer networks is known as churn.

There are different types of events that generate churn in a peer-to-peer network. One is nodes that are joining the network or parting from it. When these events are planned, i.e., the node knows about the event in advance and may give warning to the network, we call them *voluntary*. A voluntary parting is thus when a node parts from the network in an organized way. Joins are always voluntary.

Other types of events are node and network failures. Failing nodes have no time to hand over data to other nodes or even tell the other nodes that it is failing. Data that are stored in a failing node are therefore lost until the node connects to the network again. A node may also be disconnected if its network link is disabled. In some cases a disabled network link may split the network into partitions, each partition fully functional, but with a reduced data set. A node that parts the network due to a node or network failure, parts *involuntarily*.

Due to churn, the network is split into an active network $G_a = \langle V_a, E_a \rangle$ and an inactive network $G_i = \langle V_i, E_i \rangle$. In the case of network partitioning into p partitions, there are p active networks $G_{a_1} \cdots G_{a_p}$ and one inactive network. The inactive network represents resources that could become available, but at the moment are inaccessible. The active and inactive networks are non-overlapping and $G_a \cup G_i = G$.

A.3.2 Processing of Aggregation Queries

The most practical solution to aggregation queries in a P2P system with focus on distributed processing is to use a reduction tree, as illustrated in Fig. A.1. The nodes at the leaves of the tree start aggregating over their local database, and each leaf node sends its *partial aggregates*.

A partial aggregate is the information sufficient for creating a global aggregate value based on partial results from a number of sources. For example, assuming a grouped aggregate query for finding the average value of a column `c` in each group `g` of the relation `R`, i.e.:

```
SELECT g, AVG(c) FROM R GROUP BY g;
```

An example of a partial aggregate in this case is a 3-ary tuple consisting of a group identifier (a value from the `g` column of `R`), the sum so far for tuples within the group, and the number of tuples that have so far contributed to the result in the group.

An intermediate level node gathers partial aggregates from its children and merges these results with the result from the local database. The result of the merge is sent upwards in the tree to the parent node. This passing of partial aggregates continues all the way to the root node, which merges and evaluates the partial aggregates to get the final result of the query.

A.3.3 Causes for Data Loss

Queries can only access nodes in V_a, the active network. Data in nodes in V_i are inaccessible. If a node $v \in V_i$ remains inaccessible throughout the query, the only way

data stored in v may be accessible is through a replica stored at another node in V_a. However, v may suddenly join the network and make its data accessible again. The opposite may also happen. A node in V_a may part the network, voluntarily or involuntarily, during query processing. Depending on whether it has processed the query or not, its data may also be inaccessible to the query.

There are three events that may occur during query processing that can affect the total database $\mathcal{B}(V_a)$ of the active network: a node may join the network, a node may part voluntarily, and, finally, a node may part involuntarily.

When joining a network, v may bring new data to the network. If $\mathcal{B}(v) = \emptyset$, $\mathcal{B}(V_a) \cup \mathcal{B}(v) = \mathcal{B}(V_a)$, and the result of the query should be the same as if v had not joined. Nodes with no data may occur if nodes that part voluntarily hand off their data to other nodes before parting, and when new nodes are introduced to the system. If $\mathcal{B}(v) \neq \emptyset$, the total database has changed, and the result of ongoing queries may be affected. If $\mathcal{B}(v) \cap \mathcal{B}(V_a) = \mathcal{B}(v)$, all data in v are already present in the database, and duplicate insensitive aggregation functions are not affected. Since data are never lost when nodes join, the problem is limited to informing the newly joined nodes of ongoing queries and let them take part in processing these.

Nodes that part voluntarily may hand off data to other nodes before they part. This would be the natural behavior in a peer-to-peer database system. In file sharing systems nodes usually take data with them when they part, but in these systems data are usually heavily replicated, so the total database of files is not affected. However, aggregates, e.g., count of nodes that contain certain files, may change.

When nodes part involuntarily, data are lost if it is not replicated on other nodes still in V_a. Also, due to the use of reduction trees, failing nodes may contain aggregated data from other nodes. These *shadow nodes* are part of the active network, but due to the failure of another node, their data are lost to the querying process.

A.3.4 Importance of Nodes

To investigate the consequences of involuntary parting and shadow nodes, we introduce the concept of *importance* of a node. The importance of a node v is the amount of the total database v is responsible for. In a reduction tree, leaf nodes are only responsible for their own data, so if v_l is a leaf node, its importance is

$$\mathcal{I}(v_l) = \frac{|\mathcal{B}(v_l)|}{|\mathcal{B}(V_a)|}. \tag{A.1}$$

Its parent node, v_i, is an internal node with C children, $v_1, v_2, \ldots, v_C$, and its importance is

$$\mathcal{I}(v_i) = \frac{|\mathcal{B}(v_i)|}{|\mathcal{B}(V_a)|} + \sum_{c=1}^{C} \mathcal{I}(v_c). \tag{A.2}$$

The root node is in the end responsible for the whole database, making its importance

$$\mathcal{I}(v_{root}) = \frac{|\mathcal{B}(V_a)|}{|\mathcal{B}(V_a)|} = 1. \tag{A.3}$$

This is natural, since if the root node fails, all data are in its shadow, and the whole query result is lost.

In a DHT we assume a uniform distribution of tuples, so the size of the local database and hence the importance of each leaf node is the same. The formulas for importance are simplified and become

$$\mathcal{I}(v_l) = \frac{1}{|V_a|}, \tag{A.4}$$

$$\mathcal{I}(v_i) = \frac{1}{|V_a|} + \sum_{c=1}^{C} \mathcal{I}(v_c), \tag{A.5}$$

$$\mathcal{I}(v_{root}) = \frac{|V_a|}{|V_a|} = 1. \tag{A.6}$$

Generally, the importance of a node at depth h in an aggregation tree of height H is

$$\mathcal{I}_h = \frac{k^{H-h+1} - 1}{k - 1} \cdot \mathcal{I}(v_l), \tag{A.7}$$

where k is the degree, i.e., the number of child nodes at each level.

A.3.5 Expected Data Loss

Using the notion of importance, we can calculate the expected data loss caused by a single node failure. The data loss consists of both the local database of the failing node and of all nodes in its shadow, which is summed up in the importance number for that node.

In a full, perfectly balanced reduction tree where each node is of degree k, the probability of a random failing node to be a node at depth h, is

$$\Pr(h) = \frac{k^h}{|V_a|}. \tag{A.8}$$

The expected data loss caused by a single node failure is

$$\mathcal{L}_{\mathrm{H}} = \sum_{h=0}^{H} \Pr(h) \cdot \mathcal{I}_h. \tag{A.9}$$

A.3.6 Accuracy

The expected data loss, $\mathcal{L}$, is closely related to the accuracy of the query result, but accuracy depends not only on how much data is lost, but also on which data are lost. Some tuples may be more important than others, and some queries and aggregation functions can tolerate more data loss than others. One way of measuring accuracy is to look at the distance from the ideal result, i.e., the result of the query if the system was not subject to churn during query processing.

For the aggregation functions *count*, *sum* and *avg*, we simply define the distance function as

$$d_{count}(r, r_i) = d_{sum}(r, r_i) = d_{avg}(r, r_i) = \frac{r - r_i}{r_i}, \tag{A.10}$$

i.e., the percentage of deviation from the ideal result.

The *min* and *max* functions should behave similarly, and the distance between the actual and ideal result should be comparable between these two functions. The definition of distance given for *count*, *sum* and *avg* will result in large distances for small deviations from the ideal answer of the *min* function, while the same deviation will result in a short distance for the *max* function. By defining the distance function as

$$d_{min}(r, r_i) = d_{max}(r, r_i) = \frac{r - r_i}{|D_{value}|}, \tag{A.11}$$

where D_{value} is the domain of the value attribute, the distance measure should be comparable for these two functions.

The definition of the ideal result, however, is not so straight forward. In a system without churn, all nodes would be connected and all data would be present in the system at all times. Given a network $G = \langle V, E \rangle$, the ideal result of a query Q can be defined as the result of executing Q on all data residing on nodes in V, i.e., the result of Q executed on all data in the total network.

Another definition may be to consider the active network $G_a = \langle V_a, E_a \rangle$ at a given time, e.g., at the start of query execution. The ideal result is then defined as the result of executing Q on all data residing on nodes in V_a at that time.

Yet another approach is to look at the nodes $V_q \subseteq V_a$ that have actually received the query. The ideal result is then the result one would get if none of V_q fails during query execution.

Whichever definition of ideal result is chosen, it should be possible to calculate the expected accuracy under a certain churn level based on the formulas for expected data loss given in A.3.5. In these calculations one should notice that, e.g., the *min* and *max* functions are very sensitive to loss of tuples with extreme values, but tolerate much more loss of other tuples, so the distribution of values should be taken into account.

A.4 Fighting Data Loss

In this section we first look at the current method of preventing data loss on node failure, i.e., replication. We then present a two new approaches to increasing accuracy: 1) importance-based replication (IB-replication) and 2) increasing node degree.

A.4.1 Replication

The current approach to fighting data loss (as suggested by, e.g., [5] and [7]) is replication. This is easily done by creating several independent reduction trees, replicating the whole aggregation process. To run a parallel aggregation processes a complete trees have to be created, as illustrated in Fig. A.2(b) for $a = 2$. The

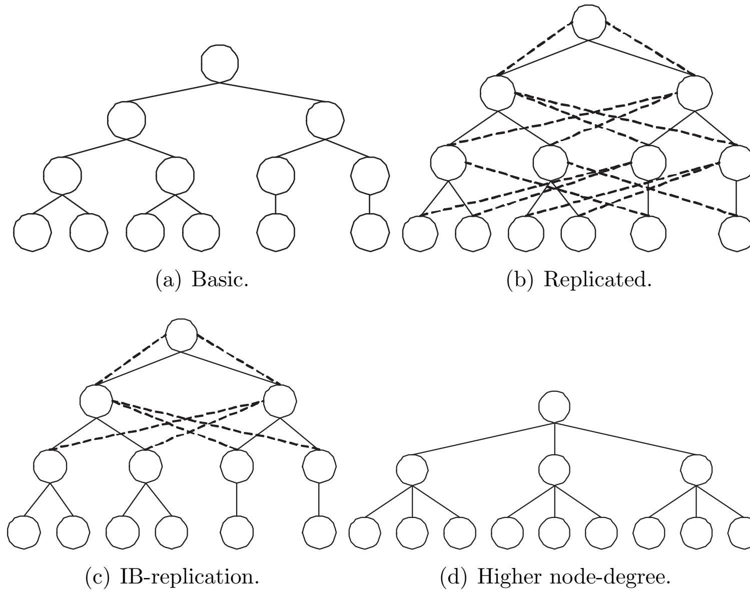


Figure A.2: Variants of aggregation trees. In the case of replication, solid lines denote one of the replica trees, and the dashed lines the other replica. Note that in general, a node participating in replica trees do not have to be in the same part of the tree in the different trees (as is the case in the figures above), it can for example be leaf node in one tree but function as a higher-level node in a replica tree. Note also that in practice, the node degrees of the aggregation trees will be much higher than in the figures.

query is executed in each of these trees. Different trees can be constructed from the routing path by selecting different destination identifiers for the tree generation message.

When aggregation is done, the algorithm is presented with a set of a complete aggregation results from which it may select one. However, this still leaves room for some unnecessary data loss, as the results for one group may be better in one replica, while the results for another group is better in another replica.

The solution is to pick the best results from each replica of the final result. In general, a result is better than another if it is based on the information from more tuples, i.e., the result with the highest count is the best. This selection of the best result may be done on the final result as a whole, but also on individual groups in the results. For each group, the algorithm selects the result from the replica with the best result.

The best result of the *count* function is the maximum result among the replicas. Similarly the best result for the *sum* function over positive integers is the highest sum. The *max* and *min* aggregation functions behave similarly, choosing the maximum and minimum among the replicas for each group. When computing the *avg* function, it is hard to tell which result is the best, but again, choosing the result

with the highest tuple count will statistically be the better choice.

The major drawback with replication is the increased number of messages sent. An a-replication results in a times the communication costs of the original algorithm. While even doubling the costs is expensive, selecting $a = 3$ or $a = 4$ would seriously degrade the scalability of the system.

A.4.2 Importance-based Replication

As noted above, replication of the complete tree results in considerably more expensive query execution. This cost may be decreased by only replicating nodes of a certain importance, with respect to Equation (A.7). This would result in a replication of the nodes that will have a big impact on the system if they fail, while the large number of lesser important nodes are kept at $a = 1$, i.e., unreplicated. Such a tree is illustrated in Fig. A.2(c). Note that in Fig. A.2(c) replication is only omitted at the lowest level, while in a higher tree more than one level might be without replication.

Importance-based replication requires the tree generation algorithm to know at which level of the hierarchy the node is placed. This information could be included in the parent node's response to the tree generation message. The replication scheme could be arranged to have several levels of replication, e.g., three copies of the immediate children of the root, two copies of their children and only one copy of other nodes. Since most nodes are placed at the leaf level or the level above, this would greatly reduce the cost of replication while still replicating the most important nodes. I.e., the result is no significant extra cost (the replicated communication and processing at upper part is very low compared to the total number of nodes in the tree participating in the processing), but accuracy comparable to full replication.

A.4.3 Increasing Node Degree

Replication is prohibitively costly for large systems, and other approaches should be followed if possible. Based on the formulas in Section A.3.4, we propose that more attention is paid to other parameters, especially the degree of nodes. A tree with larger node-degree is illustrated in Fig. A.2(d).

Using trees based on routing paths, the degree is decided by the size of the routing tables and the routing strategy used by the DHT algorithm. A large routing table that allows the node to do long jumps in the DHT space will result in a broad tree with few tiers, while a small routing table or an algorithm that only does short jumps will result in a deeper tree.

This hierarchy results in low communication costs, but the drawback is that nodes become more valuable towards the top. If one of the higher-level nodes fail, a lot of data will be lost. This effect is documented by Li et al. [7].

The importance of a node depends not only on its level in the hierarchy, but also on the degree. This is evident in Equations (A.2), (A.5) and (A.7). If the tree is broad, each of the higher level nodes are less important. A pathological case is when the degree is 1, i.e., the hierarchy is a linked list where the importance is increased

by one for each level of the hierarchy. If one node is lost, all information from nodes below it in the hierarchy is lost. Assuming a uniform failure model, this hierarchy would on average lose half of the information each time a node fails.

If the hierarchy is a binary tree, the two nodes below the coordinator node are each responsible for one half of the information in the tree. Instead of losing half the information on average, it is the worst case loss. If the degree is three or four, the expected data loss is still lower. In the extreme, the degree is equal to the number of nodes, in which case the distributed aggregation degenerates into centralized aggregation.

If routing paths are used to build the hierarchy, the degree and depth of the hierarchy are decided by DHT algorithm parameters. The size of the routing table is one of the factors that decide how many hops a message needs to get to the root, and therefore also the depth of the tree. The properties of the hierarchy are also dependent on the routing policy of the network, e.g., if the DHT algorithm always routes messages in a greedy manner or if it takes other aspects into consideration. The degree of nodes could be changed either by modifying the routing principles of the algorithm, or by disconnecting the tree generation from the routing path.

In [7], Li et al. describe a tree construction protocol that takes the degree of internal nodes as a parameter. Such functions could be used to generate trees independent of routing paths, thus deciding in each case the desired degree of the tree.

The tree generated by existing algorithms can be estimated by considering the height of the tree to be the maximum length of the routing path, i.e., the maximum number of hops when doing a lookup. Assuming that all nodes have the same degree and that the tree is completely balanced, the degree can be calculated. The connection between number of nodes, N, degree, k, and height h of the tree is given by

$$N = \frac{k^{h+1} - 1}{k - 1}.$$

The Chord algorithm has a high-probability upper bond of $O(\frac{1}{2}\log_2 N)$, where N is the number of nodes. Experiments show that the routing path length for a Chord network of 10,000 nodes varies from 2 to 11 [15], with an average of approximately 5. Assuming that all nodes have the same degree, and that the tree is completely balanced and full, a network of 10,000 nodes with maximum path length 5 has a degree of approximately 6.

In CAN, the degree depends on the number of dimensions. Experiments in [10] show that the number of hops in a 4-dimensional CAN of approximately 130,000 nodes, the path length is approximately 5, which should give a degree of approximately 10.

These low numbers indicate that more attention should be paid to the degree of nodes in the reduction tree. From Section A.3.4 we see that the degree is directly related to the importance of nodes, and hence, the expected data loss.

A.5 Experiments

In this section we compare full replication and varying node-degrees by simulating aggregation by reduction trees in a DHT. Results from importance-based replications are omitted as their performance will be close to full replication (although at a very much lower cost).

A.5.1 Network Model

The simulated network system is a DHT network experiencing different churn levels. The focus is on the results of the queries, not on the number of messages sent between nodes or other network metrics. This allows for some simplifications in the network model.

The first assumption that is made, is that nodes that leave the network without failure, i.e., in a planned, organized way, will hand over data and ongoing queries to other nodes before disconnecting. This can be done either by transferring data and queries to other nodes, or by entering a state where the node completes all ongoing queries, but does not accept new queries. When a node no longer has active queries or data, it can leave the network. This assumption is one on the behavior of the software system, not of the network structure.

Based on this assumption, the network (without node failures) is modeled as of constant size, i.e., the number of nodes joining is equal to the number of nodes leaving the network.

Node failures are assumed to occur after the node has received all messages, but before it sends any messages. This means that failed nodes are not discovered by the network, and that all messages to failed nodes are lost. The justification for this assumption is that nodes that discover failed nodes can take actions to overcome this problem, e.g., update its routing tables and route messages through a different node. There are still situations where node failure will not be detected, e.g., if a node fails during phase one of the repartitioning algorithm after it has received all tuples from one node.

Since some cases of node failure are supposed to be discovered and handled otherwise, the number of node failures for the simulations should be lower than in the corresponding real-world situation.

The simulations are run on a network of 10,000 nodes. 10% of the nodes fail during query processing. This is a fairly high number, chosen to show how the algorithms perform under the bad conditions.

A.5.2 Data and Query Model

The data model is that of a relational database consisting of a single 100,000 tuple relation which is distributed over all nodes. Since the network is based on a DHT, a uniform distribution of tuples is assumed.

The aggregation functions studied in the experiments are the standard SQL functions *sum*, *count*, *avg*, *min* and *max*.

The tuples have 3 attributes: *key*, *group* and *value*. The *key* attribute is a unique value which is used as the primary key for the tuple. When aggregating, the results are grouped by the *group* attribute, and the *value* attribute is used as parameter to the aggregation function. All values are positive integers. The *value* and *group* attributes are chosen randomly from their domains, using a uniform distribution.

A.5.3 Algorithm Implementation

The generated reduction trees are completely random, and not based on any DHT. This design choice allows the simulator to construct trees with different properties, so that the effect of changing the degree can be studied.

When doing replication, all replicas of the result are examined to pick out the best result for each function over each group, as described in Section A.4.1. Replication is done by generating new reduction trees.

A.5.4 Metrics

The results are compared to an ideal result aggregated over $\mathcal{B}(V_a)$, and the accuracy of the query result is calculated using the formulas defined above. As a result, the accuracy of *min* and *max* cannot be compared directly to the accuracy of the other aggregation functions.

Each query is run 10,000 times, and the mean and inter-quartile range is used when discussing the results of the experiments.

A.5.5 Results

First we look at the effect of replication. Fig. A.3 shows the accuracy for *avg*, *count* and *max* functions. The results for *sum* and *min* functions are similar to those of the *count* and *max* functions, respectively, and are not reproduced in the figure. The figure shows the mean value, and lines extend to the first and third quartile to show distribution density.

We see that the results of the *count* function are more inaccurate than those of *avg* and *max*. For *avg* this is the result of the DHT distributing values randomly. The loss of one node does not result in systematic data loss, so the average value is not severely affected by data loss.

The *max* function depends on a specific value being present in the accessible dataset. If this value is not lost, the function will return the correct answer. If there are more than one tuple with this value, the function is even more resilient to data loss.

The *count* function, however, depends on every tuple being present, and is thus much more vulnerable in case of node failures. Unlike the other functions, which come much closer to an accurate result even when not replicated, *count* (and *sum*) clearly show improvement when replicated.

From the results, we see that the accuracy can be increased somewhat by replicating the process, but that there is little to gain by increasing to three or four

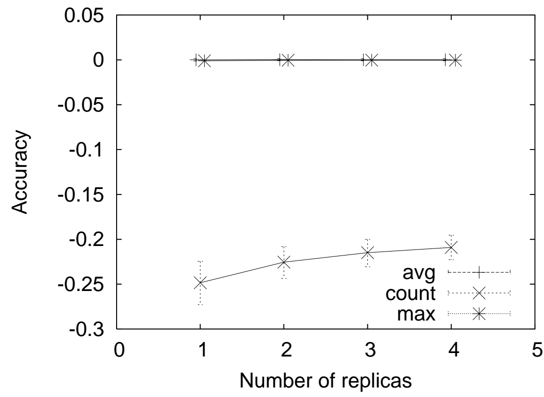


Figure A.3: Accuracy of aggregation functions with different number of replicas.

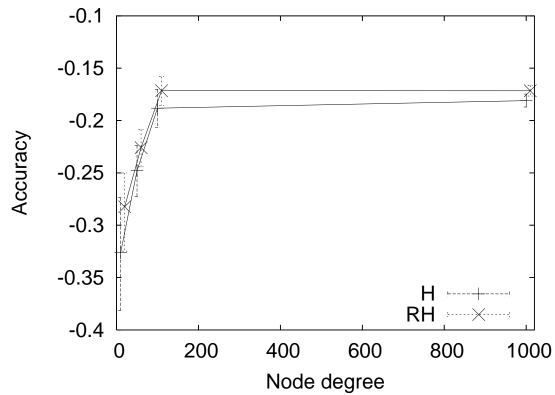


Figure A.4: Accuracy of the count function with different node degrees.

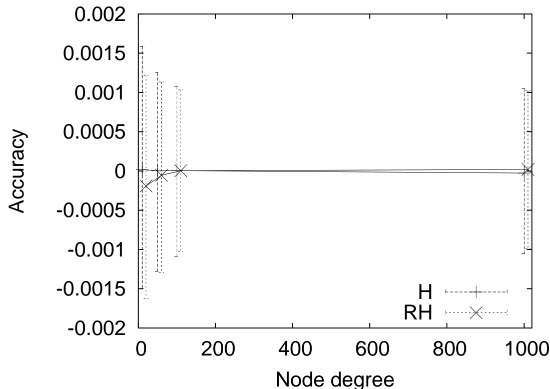


Figure A.5: Accuracy of the avg function with different node degrees.

complete processes. Replication is a costly solution, and if there is little to gain in terms of accuracy, it may not be worth the cost. Also, the *avg*, *min* and *max* functions prove to be quite accurate to start with, whereas the *count* and *sum* functions show that some method to fight data loss is needed. In the rest of the experiments, only single replication is used, so the results compared are from simple hierarchical aggregation (H) and replicated hierarchical aggregation (RH).

Fig. A.4 shows the results of varying the degree of nodes in the reduction tree from 10 to 1,000 (using steps 10; 50; 100; 1,000). We can see that the accuracy of *count* queries climb quite steeply from 10 to 100, i.e., from 0.1% to 1% of the number of nodes in the system, but that accuracy does not increase much beyond this number. The replicated algorithm (RH) performs better than the non-replicated algorithm (H), but for low node degrees, there is more to gain by increasing the degree than by replicating.

Due to implementation details, the replicated algorithm actually performs worse for low node degrees when computing the average, as can be seen in Fig. A.5. The reason for this peculiar result, is that the simulator computes the result of *avg* from the results of *sum* and *count*.

The algorithm chooses the best result for *count* and *sum* separately, and only computes *avg* in the end. The result is that the two components of the partial aggregate for *avg* are chosen from different replicas which generally do not contain the same tuples, and the result of the query becomes more inaccurate. This shows that it is important to choose all elements of the partial aggregates from the same replica, even though the single parts may be more accurate by themselves in different replicas.

We also see that there is little to gain by increasing the degree when computing *avg* queries. The distribution becomes somewhat denser, but not much.

When computing the maximum value, the results in Fig. A.6 show that the node degree is important, but that there is more to get from simple replication. For the parameters used in our simulations it is always better to replicate than to increase the degree of internal nodes.

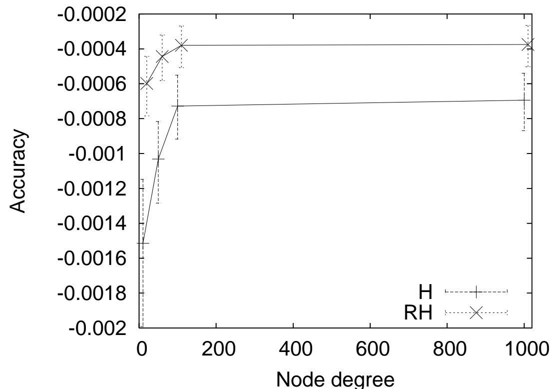


Figure A.6: Accuracy of the max function with different node degrees.

The results from the experiments show that the different aggregation functions react differently to varying node degrees. The *count* function has much to gain from increasing the node degree before replicating the process, but *max* gains more from replication than from increasing the degree. The results also show that the two techniques can be combined to increase accuracy further. The *avg* function is quite accurate to begin with, and does not seem to react much to either method.

If we compare the results to the estimated node degrees of trees based on routing paths in Chord and CAN given in Section A.4.3, we see that the simulation results indicate that trees based on current DHT implementations are too narrow, and that accuracy could be increased by generating broader trees.

A.6 Conclusion and Future Work

We have performed a formal study of data loss in aggregation queries and described the different events that may affect the result of such queries. The focus has been on the uncontrollable events, i.e., node and network failure, and how algorithms can be adapted to provide accurate answers in a setting where node failures are common.

Based on this analysis, we have proposed new approaches to increasing accuracy. Instead of just replicating the whole aggregation process, which until now has been the suggested solution, we proposed two alternatives based on importance-based replication and the degree of internal nodes in aggregation trees.

Our experiments showed that these new approaches in some cases may be more efficient in increasing accuracy than the costly replication, and also that these two methods may be combined to increase accuracy further. The simulations also indicate that there is much to gain from increasing the node degree from that of current implementations.

Several open problems remain. The query processor should be able to use statistics to predict which algorithm and which parameters would suit the query best. This could be combined with a requested level of accuracy to find the most efficient

aggregation method to achieve the requested accuracy.

The data and accuracy loss of other relational operations should also be studied, so that queries do not suffer unnecessarily from data loss. When planning query execution, the methods chosen for each operation should be selected to achieve the requested accuracy. This requires a formal study of data loss and functions for predicting accuracy of relational operations.

Bibliography

[1] B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki. Approximating aggregation queries in peer-to-peer networks. In *Proceedings of ICDE'2006*, 2006.

[2] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proceedings of SIGMOD'2004*, 2004.

[3] A. Demers, J. Gehrke, R. Rajaraman, N. Trigoni, and Y. Yao. The Cougar project: A work-in-progress report. *SIGMOD Rec.*, 32(4):53–59, 2003.

[4] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[5] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of PIER: an internet-scale query processor. In *Proceedings of CIDR*, pages 28–43, 2005.

[6] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of VLDB'2003*, 2003.

[7] J. Li, K. Sollins, and D.-Y. Lim. Implementing aggregation and broadcast over distributed hash tables. *SIGCOMM Comput. Commun. Rev.*, 35(1):81–92, 2005.

[8] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[9] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of ICDE'2003*, 2003.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM'01*, 2001.

[11] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[12] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware'2001*, 2001.

[13] S. Roy, S. Setia, and S. Jajodia. Attack-resilient hierarchical data aggregation in sensor networks. In *Proceedings of SASN*, 2006.

[14] N. Ryeng and K. Nørvåg. Accuracy of aggregation in peer-to-peer DBMSs. In *Poster presented at DBISP2P'2007*, 2007.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM'01*, 2001.

[16] P. Valduriez and E. Pacitti. Data management in large-scale P2P systems. In *Proceedings of VECPAR'2004*, 2004.

[17] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of SIGCOMM '04*, 2004.

Paper B

RIPPNET: Efficient Range Indexing in Peer-to-Peer Networks

Norvald H. Ryeng and Kjetil Nørvåg.
In *Proceedings of ICDIM*, 2008.

Abstract

Write-heavy applications present a challenge to peer-to-peer indexing methods which need to update the index for each write operation. The costs incurred when the distributed index is updated becomes a bottleneck. Current distributed indexing methods are designed for indexing and retrieving single tuples, giving a very high update cost. In this paper we present a new approach to efficient peer-to-peer range indexing that employs indexing of ranges to reduce average update costs as well as providing efficient data localization and decoupling from data placement policies. Based on results from experiments, we demonstrate the applicability and significantly reduced update cost of the new approach.

B.1 Introduction

Large, distributed simulation processes can produce a lot of data. In some simulations the number of write operations are much higher than the number of read operations, and data is more often read from local storage than from other nodes. A distributed relational database management system (RDBMS) for large simulations, e.g., in a computational grid, must be able to efficiently perform writes, while at the same time allowing global lookups. The distributed, self-organizing nature of peer-to-peer systems provides a good basis for building such RDBMSs.

Existing peer-to-peer RDBMSs use tuple based indexing, which requires the distributed index to be updated every time a tuple is inserted, updated or deleted. For write-heavy systems, the cost of updating the distributed index is a bottleneck. In this paper we present RIPPNET, an indexing method for peer-to-peer RDBMSs that instead indexes ranges of tuples. The local database at each node is divided into ranges that are registered in the distributed index. By indexing ranges instead of tuples, the number of messages used to keep the index up to date is reduced, while still allowing for data localization queries.

The common indexing method for structured systems is to use a distributed hash table (DHT) that indexes tuples. Since the tuples are inserted into the DHT, the DHT enforces a certain data placement based on the hashing function. Our range index is orthogonal to any data placement policies. This is important in a simulation setting where each node mostly accesses the data produced locally, but occasionally needs to access data from other nodes. If the index structure dictates a data placement policy that is incompatible with the pattern produced by the simulation, the result is a lot of network traffic to write or read data. By decoupling indexing from data placement, many reads can be made local.

Thus the main contribution of this paper is an indexing method for peer-to-peer RDBMSs that:

- indexes ranges of tuples,
- significantly reduces costs for write-heavy systems, and
- is decoupled from data placement policies.

The described indexing method is implemented in a simulator and results from experiments show that update costs are significantly reduced.

In this paper we start by reviewing related work in Section B.2 before we present some preliminaries in Section B.3. Our new range indexing method is presented in Section B.4, and extensions to the basic method are described in Section B.5. Finally, we evaluate our approach and achieved results in Section B.6 and conclude the paper in Section B.7.

B.2 Related Work

Current systems typically use DHTs such as Kademlia [12], Chord [18], CAN [15], Pastry [16] or Tapestry [10] to store tuples or tuple indices. These systems support only exact match lookups. More advanced queries, such as range and cover queries, are blocked by the hashing function, which destroys the data ordering.

Two techniques for performing range queries in peer-to-peer systems are discussed in [21]. The first is simply to give the same hash key to values within a range, thereby reducing the number of different hash keys and DHT lookups needed. The other technique is to create a multicast group for each range.

Gupta et al. [8] present a range selection technique for DHTs based on locality sensitive hashing (LSH). The method locates data in $O(\log N)$ hops in a N-node network. However, the suggested methods only give approximate answers. HotRoD [14] uses a combination of locality-preserving hashing function and replication to support range queries in a DHT-based system. In [1] and [4] data are distributes contiguously into a DHT-like ring, but without using the hashing function, relying on load balancing algorithms to maintain fairness in case of data skew. GChord [23] utilizes Gray coding to support range queries. Consecutive values differ in only one bit, and this fact is used to forward queries through the Chord network.

Another approach is to organize the data into a tree structure. This class of systems include the Distributed Segment Tree [22], the Range Search Tree [7], BATON [11], P-Grid [2] and P-Tree [5]. A similar approach is search tries stored in DHTs [19].

The Distributed Segment Tree is a binary tree that can handle both range and cover queries. Unfortunately, nodes in all levels must be updated when a tuple is inserted or deleted. To reduce the load of higher-level nodes, these nodes can decide to drop tuples belonging to their children, but still the message has to be sent. In the Range Search Tree, each node of the tree consists of a load balanced set of physical nodes.

P-Tree is based on B^{+}-trees and uses Chord as the underlying DHT. Tuples are stored in leaf nodes, which constitute the Chord network. P-Grid places data in a binary prefix tree where each node maintains references to other nodes with the same prefix. Both P-Tree and P-Grid have worst case scenarios where the tree degenerates into a linked list. BATON overcomes this limitation by self-adjusting to data skew.

SkipIndex [20], SkipNet [9] and Skip Graphs [3] and ZNet [17] are based on Skip Lists, a tree of linked lists, where the list at level 0 is a linked list of all nodes. Higher

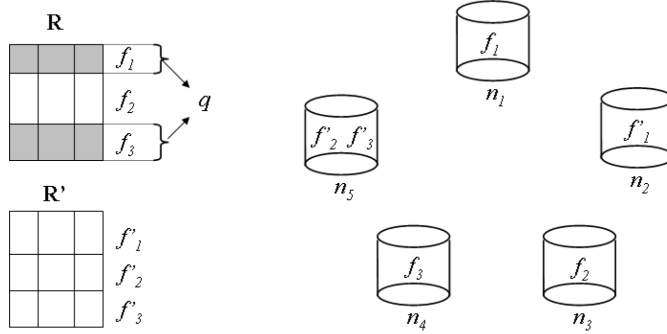


Figure B.1: An example system.

level lists are increasingly sparse. Using these lists, range queries can be supported.

The RangeGuard system [13] uses a set of supernodes to allow for range queries in a DHT. The supernodes form a ring, each supernode taking responsibility for one range of the value space.

Common to all these range search strategies, are that they are based on tuple indexing. The cost of updates vary between the different structures, but all have to perform some work on every tuple update. Our approach is to store ranges in the index, thereby reducing the number of update messages sent to the index.

B.3 Preliminaries

In this section we first present our system model and then the problem of distributed data localization.

B.3.1 System Model

In our model of a peer-to-peer DBMS, tuples are stored in horizontal fragments of relations. Relations are fragmented based on some fragmentation rule that may be local to a node or system-wide. A node may store more than one fragment of a relation, and these fragments need not be consecutive.

In the example shown in Figure B.1, there are five nodes, $n_1, n_2, \ldots n_5$. Nodes n_1, n_3 and n_4 each store a fragment, numbered f_1, f_2 and f_3, respectively, of relation $\mathbf{R} = \bigcup_{\forall i} f_i$. Node n_2 stores one fragment, f'_1, and node n_5 stores two fragments, f'_2 and f'_3, of relation $\mathbf{R}' = \bigcup_{\forall i} f'_i$.

It is our assumption that the data distribution is not uniform, at least not within a single node. Skewed data sets are common in real life applications, such as the distribution of names, and in our model we assume that this is generally the case.

For each fragment there exists a minimum and maximum allowable value for an attribute of a tuple in that fragment. Simple fragmentation rules may fragment based on only one attribute, e.g., separating the relation into fixed steps of the

fragmentation attribute. This would limit the value of the fragmentation attribute in each fragment, but leave the other attributes limited only by the limits of the data type. The allowed range of the fragmentation attribute may be significantly larger than the range that is actually used.

B.3.2 Data Localization

The localization step in a distributed query processor needs to translate a query on global relations $\mathbf{R}$ and $\mathbf{R}'$ to a query on the physical fragments, $f_1, \ldots, f_3$ and $f'_1, \ldots, f'_3$, of the relations. Not all fragments are needed for all queries. If the query is for all tuples where an attribute is within some range, only fragments containing tuples within that range are needed. We call those fragments *relevant* to the query. Other fragments of those queries are *irrelevant* to the query. In the example in Figure B.1, query q separates the fragments of $\mathbf{R}$ into the relevant fragments, $\mathbf{R}_q = f_1 \cup f_3$, and the irrelevant fragments $\mathbf{R}_{\overline{q}} = f_2$. This concept of relevance is extended to nodes, such that relevant nodes contain at least one relevant fragment, and irrelevant nodes contain no relevant fragments. Given a query, q, we divide the set of nodes, $\mathbf{N}$, into those relevant to the query, $\mathbf{N}_q$, and those irrelevant to the query, $\mathbf{N}_{\overline{q}}$.

If the system is very small, it is more efficient to broadcast the query to all nodes, relevant or not, instead of first identifying relevant nodes and then send the query only to these nodes. The disadvantage of broadcasts increases with $\frac{|\mathbf{N}_{\overline{q}}|}{|\mathbf{N}_q|}$, the ratio of irrelevant nodes to relevant nodes. At some point, the cost of broadcasts exceeds the cost of identifying relevant nodes. Exactly when this occurs, depends on the cost of identifying relevant nodes.

When the system is very small, a complete index of all fragments can be stored on all nodes. As the system grows larger, this becomes infeasible. For each node to have complete knowledge of all fragments, even of all nodes, requires too much communication and state information. Currently most systems use DHTs, both to store tuples and to create distributed indices. The DHT allows all nodes to look up and retrieve data from all other nodes, with only a small amount of state information on each node.

One disadvantage of DHTs is that they only allow exact match lookups. If the relevant fragments for a query are all those within a range, we have to look up every single possible value within that range. The total cost of a query for a range using this method depends on the cost of a single exact match lookup and the width of the range in question. E.g., if there are only two possible values within the range, the two exact match queries required will not be a very costly range search. However, if the range is wide, the cost of looking up every possible value within the range makes it infeasible.

Efficient data localization is not without cost. For indices to be useful, they must be kept up-to-date. Maintaining an index of all tuples is costly. Every time a tuple is inserted in a relation in the system, a new index entry must be inserted. This detailed index is useful for some purposes, but overly detailed for data localization purposes. The data localization step only needs to find out which nodes are to be

involved in the query, not the location of every single tuple.

A simple solution is to index ranges of tuples instead of single tuples. To find relevant nodes, the data localization step has to find all ranges that overlap the range requested by the query. For each range stored in the index, a node identifier is stored. A range query, q, will be answered by a set of relevant ranges and corresponding relevant nodes, $\mathbf{N}_q$.

B.4 Distributed Range Indexing

We propose a distributed index of ranges of tuples where each node defines ranges of tuples in its local database and stores information about these ranges in a distributed index. A query processor that needs to identify all nodes storing data within a range looks up in the index and finds all intersecting ranges.

The indexing method can be used on multidimensional data, e.g., indexing over multiple attributes of a relation. For ease of presentation, the examples will be limited to indices over one attribute.

To build, maintain and use a distributed range index, there are three main processes: range partitioning of data and building the index, maintaining the index, and searching the index. We will treat these processes separately.

B.4.1 Partitioning Local Data

Local data should be divided into ranges based on the indexing attributes. This could be done using using any clustering algorithm, or using fixed steps in the attribute domain. An incremental range partitioning method that allows for growing or shrinking of ranges is necessary. Using such a method, the algorithm would not have to rescan the whole database when a tuple is added.

Unlike many other applications, the ranges can be overlapping, i.e., a single tuple can belong to several ranges. However, index lookups will gain from having dense ranges, so adding tuples that are too distant from the rest of the range will probably not gain anything. Also, keeping the index updated when tuples are inserted, updated and deleted is easier when tuples belong to only one range.

Outliers may occur, and when deemed to far away for inclusion into one of the other ranges, they may form ranges of their own. However, since these outliers are in fact indexed as single tuples, the range partitioning algorithm should try to generate as few outliers as possible.

For each range identified, we create an index record. The only required fields of this record is the minimum and maximum values of the range and the network address of the node where this range is stored.

The index record can be further extended by storing various statistics on the range, such as the number of tuples, etc. This information is not necessary for the index to work, but may be useful to the query planner when constructing a query plan. This is discussed further in Section B.5.3.

The data distribution on one node may look similar to that displayed in Figure B.2. This node stores data that can be partitioned into four ranges, r_1, r_2, r_3

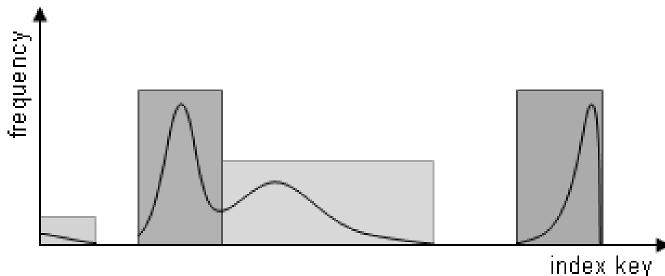


Figure B.2: Data distribution on a node.

and r_4. These ranges cover only the parts of the attribute domain where the node has data. The ranges of one node need not cover all possible values. The example node has no outliers.

If the distribution is much more uniform than in our example, one may have to resort to fixed-width partitioning to define ranges. These will probably be sparser than what is outlined above, but the indexing method can still be used.

Each node has to identify ranges in its local database before the index can be built. Later, we will see how the ranges are maintained when tuples are inserted, updated and deleted.

B.4.2 Building the Index

When each node has partitioned its data into ranges, we can build a distributed index of all ranges. When indexing over d attributes, we build a $2d$-dimensional index. The dimensions are the minimum and maximum values of each of the d attributes.

Let us consider an index over a one-dimensional attribute, e.g., an integer value. Our index would then be a two-dimensional index, consisting of the pair of minimum and maximum values for all ranges. Each range, r, would be represented as a point, $\langle r_{min}, r_{max} \rangle$, in the index space, as shown in Figure B.3.

Since the minimum value of a range is always smaller than the maximum value, the possible index records form a triangular space in the two-dimensional index. When partitioning data into ranges, some data points may be considered outliers and will be stored as single points in the index. Since the minimum and maximum values are equal for these points, they will be stored along the diagonal.

This structure can be stored in a Content Addressable Network (CAN) [15]. We have to bypass the hashing step and store the values directly in the CAN. The hashing algorithm makes sure data is evenly distributed among the nodes. When we bypass the hashing step, data will be unevenly distributed among the nodes. In particular, only the upper left triangle of a two-dimensional network will be used. In Section B.5.2 we look at how we can keep a close to uniform data distribution in the network without the hashing step.

New index records are inserted into the index in the same way as for a normal

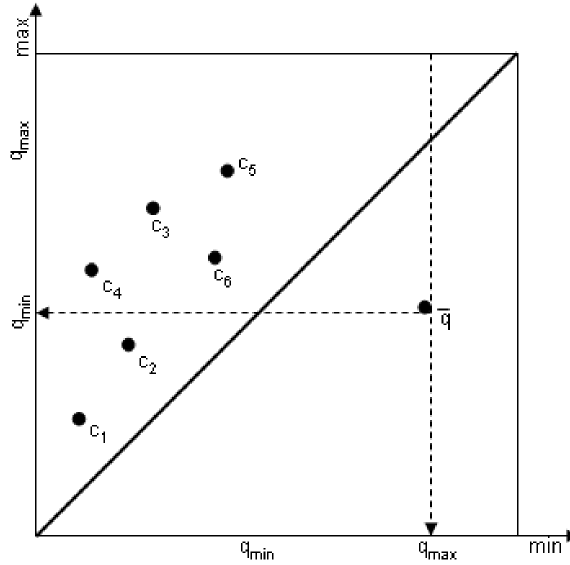


Figure B.3: Two-dimensional index of one-dimensional data.

CAN, except that the hashing step is bypassed. The index is updated when ranges in the original database changes. If a range is deleted or merged into another range, the corresponding index record is deleted. Updates and deletes are also similar to their normal CAN counterparts.

B.4.3 Maintaining the Index

Once the index has been created, it must be kept up-to-date. On inserts, updates and deletes, nodes must check if index records have to be updated.

On inserts, nodes must check if the tuples fit inside already existing ranges. If so, no further action needs to be taken. If not, it must decide if it should extend an existing range or define a new range to cover the tuples.

On deletes, nodes must check if it is possible to shrink the range the deleted tuples belong to. Too wide ranges does not affect the result, but it affects the performance of the index. Index records that describe a too wide range will result in more messages sent to nodes that do not store data within the requested range. If ranges are overlapping, there may be more than one range to check.

Updates are treated as a combination of inserts and deletes, and the corresponding actions must be taken.

To avoid having to look up in the distributed index for every insert, update and delete, nodes should store information about local ranges. This local information should be enough to decide when to extend or shrink existing ranges and when to create new.

If extra statistics are stored in the index records, care must be taken to also up-

date this. Statistics updates may occur more often than ranges have to be extended or shrunk, so for this reason, exact count and similar statistics should be avoided.

The result of indexing ranges is that the index does not have to be updated for every insert, update and delete of database tuples. The exact savings in communications costs depends on the data set and the frequency of such requests.

B.4.4 Index Lookups

The index allows for two types of lookups: range queries and exact match queries. Range queries are used to answer queries for indexed data, while exact match queries are used when updating index records.

Range Queries

There are four different situations that may occur when comparing two ranges. In our case, we will compare an indexed range, r, with the range of the query, q. The different situations that may occur are that

- r and q overlap in the lower part of r and the upper part of q,
- r and q overlap in the lower part of q and the upper part of r,
- q is contained in r, and
- r is contained in q.

The index is queried by ranges. A query q for the range $\langle q_{min}, q_{max} \rangle$ is represented in Figure B.3 by its reverse range point, $\bar{q} = \langle q_{max}, q_{min} \rangle$, as shown in Figure B.4. All points in the area delimited by the vertical axis, the line from $\bar{q}$ perpendicular to the vertical axis and the vertical line extending from $\bar{q}$ perpendicular to the horizontal axis represent ranges that overlap, contain or are contained by the range of q.

Query execution is done by routing a message to $\bar{q}$. The node containing $\bar{q}$ then forwards the query to its neighbors within the requested range, which again forwards the query to their neighbors, etc. In this way the query propagates through the network until it reaches all index nodes potentially containing overlapping ranges.

Any node that in this process receives the query, in addition to forwarding the query to its neighbors, responds to the querying node with a list of matching ranges.

As a result of this query propagation scheme, narrow ranges will be reached before wide ranges. These narrow ranges are also more likely to contain useful tuples. As shown in Figure B.4, the first ranges that are encountered, are those that are contained within the query range, and thus guaranteed to only contain relevant data. The query then continues to partially overlapping ranges and ranges that contain the query range. The wider ranges are indexed on the last nodes to receive the query, and hence, returned last.

Cover queries are queries for regions that contain a certain point. The distributed range index can answer cover queries by answering the range query for ranges overlapping the range with minimum and maximum values equal to this point.

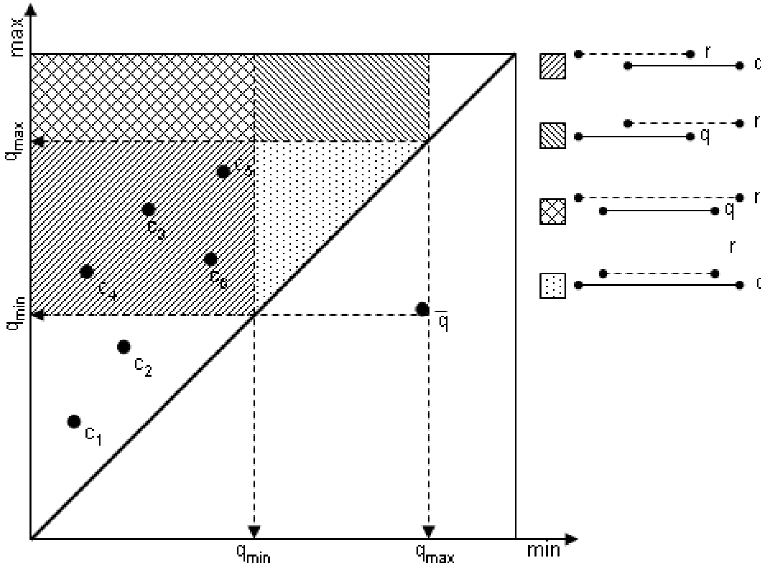


Figure B.4: Range placement.

Exact Match Queries

The index can also be queried for an exact match, e.g., when a record needs to be updated. This is done in the same way as it is done in a normal CAN. This is a query type used to maintain the index. Exact match queries for indexed values are done as cover queries, since there may be more than one range covering the requested value.

B.5 Extensions

Section B.4 presented the basic idea. In this section we present extensions to the basic idea for handling multidimensional data, load distribution and more advanced index records.

B.5.1 Multidimensional Data

The proposed range indexing technique supports multidimensional indices. Indexing d-dimensional data requires a $2d$-dimensional CAN, since for each dimension the index needs one minimum and one maximum dimension. The easiest mapping would then be to map minimum values to even numbered dimensions and maximum values to odd numbered dimensions, such that for each dimension i of the key, the index has two dimensions, $i_{min} = 2i$ and $i_{max} = 2i + 1$.

Since the dimensions are not independent, with increasing dimensions, a smaller percentage of the CAN is actually used to store data. The problem of uneven data

and computational load distribution is discussed in the next section.

B.5.2 Uniform Distribution

An unfortunate effect of bypassing the hashing step of the CAN is that the data distribution is no longer uniform. For a two-dimensional CAN, half the address space is not used by the index. Also, the computational load is not distributed evenly. In Figure B.3, the node storing the upper left corner of the CAN will be involved in every index lookup.

The naive approach to solving the data distribution problem is to swap maximum and minimum dimensions for different indices. If multiple indices are stored in the same CAN, they can use different dimensions as minimum and maximum dimensions, thereby evening out the data distribution. Also, the direction of dimensions can be switched. This does not guarantee uniform distribution, but helps in distributing data to all nodes, not only one half of them.

When dimensions are swapped and reversed, this also helps even out the differences in computational load, but still the corner nodes of a two-dimensional CAN will be more heavily loaded than other nodes. The most heavily loaded nodes could be replicated, using a round-robin algorithm to choose which replica to use for a specific index lookup.

B.5.3 Statistics in Index Records

Statistics on range size and distribution could be stored in the index record. If exact answers are not needed, this could be used to speed up aggregation queries, using the statistics to estimate the aggregate without asking the nodes where the data are stored.

The query planner can also use statistics to select first the nodes with a high density of tuples within the requested range. This could also be used to give a quick reply that gradually improves as the rest of the database is searched.

There are disadvantages to storing statistics in the index records. For the statistics to be correct, the index record must be updated more often than the range indexing mechanism requires. When adding statistical information to the index records, one must be careful not to require too frequent updates. The information in index records should change slowly and be defined as within some error margin, to avoid updating the index record for every single tuple insert, update or delete.

B.6 Experimental Evaluation

The proposed indexing technique was implemented in a CAN simulator that allowed us to experiment with different network sizes and database sizes. The proposed range indexing method is compared to a baseline method where queries are broadcast to all nodes, but only those nodes that contain relevant data reply.

B.6.1 Setup

The experiments were done using a Java-based CAN simulator extended with range query capabilities as described in Section B.4. The simulator ran one query at a time, waiting until one query finished before the next was issued.

Network Model

For each of the network sizes, the network used was the exact same for each query. Nodes joining the network were given responsibility for zones as described in [15], using a random number to find the zone to split.

The simulated networks were static. There were no nodes joining or leaving the network during simulation.

Messages were forwarded through the CAN only when doing the actual lookup. The lookup message contained a node identifier of the querying node, and this identifier was used for direct communication outside the CAN. When overlapping ranges were found, the results were returned to the querying node directly using this identifier.

Data Set

A database was created for each node. The advantages of the indexing strategy is based on principle of data locality, i.e., data on a single node tends to be similar, or clustered. For each node a set of random seed points were chosen. From these seed points data clusters were grown.

An index was made over one attribute, a positive integer. The data on each node was partitioned into ranges using the DBSCAN [6] clustering method. Outliers were inserted as ranges consisting of single tuples.

Query Model

In each experiment, the network was queried 10,000 times. The range queried was chosen randomly, but the width of the query was fixed to a certain percentage of the attribute domain.

Metrics

For each query, we measured the number of messages used to propagate the query and return a complete answer. This number includes both the propagation of the query through the CAN and the direct return messages from nodes in the index to the querying node.

The first experiment describes the update frequency. In this experiment, the probability was measured in a network of 1,000 nodes, inserting the tuples in the order they were generated. For each node it was recorded whether the total range of the node had to be updated when a new tuple was inserted.

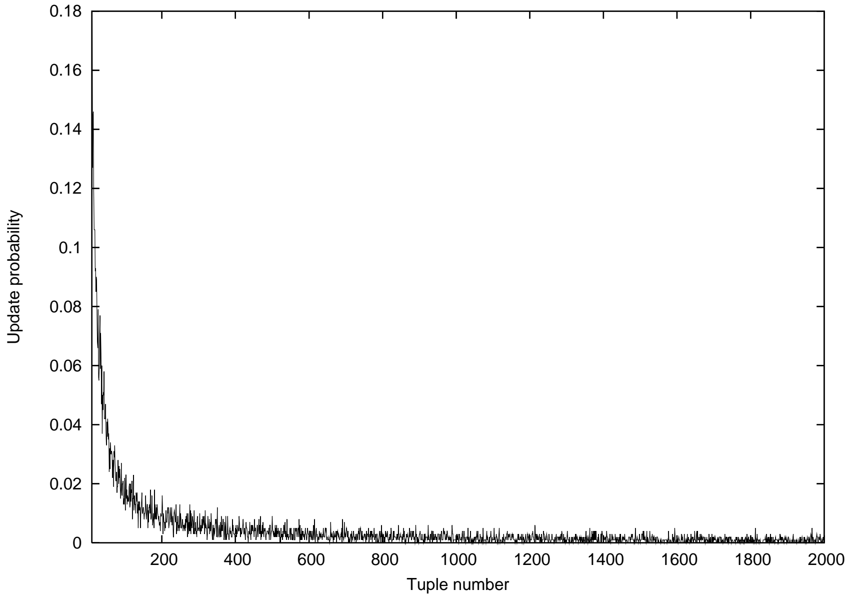


Figure B.5: Probability of index updates.

B.6.2 Results

We now describe the results of three experiments. In the first experiment, we looked at the probability of index updates during tuple inserts. Then we looked at how the cost of querying varies with network size and query range.

Update Rate

Tuple indices must be updated on every tuple insert, update and delete. When indexing ranges, the update frequency can be reduced if new tuples that are arriving fall within a range that already is registered in the index. In this experiment we looked at the first 2,000 tuples inserted into each node of a 1,000 node network. The data set was the same as used in the other experiments, where points tend to cluster around a few points. Figure B.5 shows the probability of tuples to extend the range and causing an index update.

The first 10 inserts have a very high probability of causing index updates, so they have been removed from the figure to allow us to see better what happens afterward. As we see from the figure, the probability of new tuples causing index updates is greatly reduced as the database fills up. Already after 10 inserts, the probability of causing an index update is reduced to 17.4%. After about 150 inserts, the probability of index updates is reduced to below 1%. This can be compared to the 100% probability of update in a tuple based index.

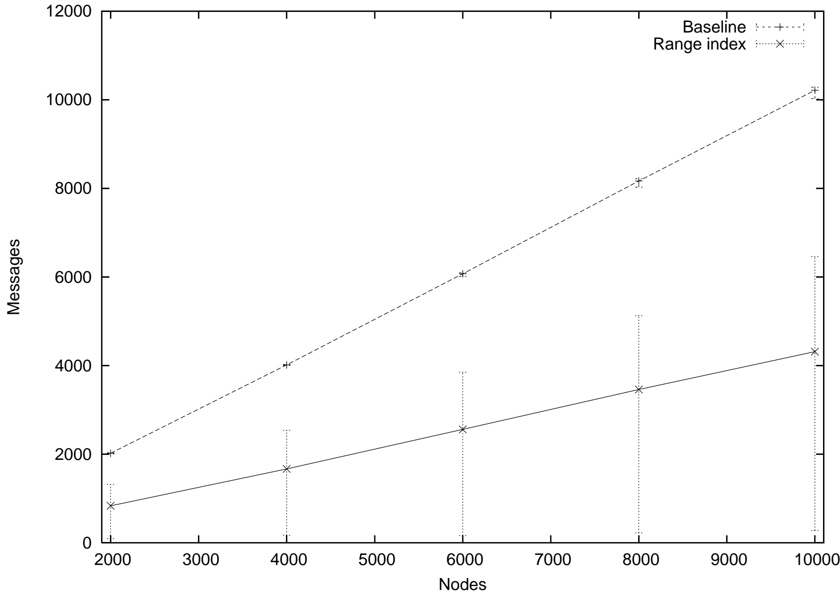


Figure B.6: Messages used in varying network sizes.

Varying Network Size

In this experiment we looked at the number of messages used to look up a range for varying networks sizes. The number of nodes in the network is the main parameter that decides the cost of a range lookup. The database used was a one-dimensional database of 10,000,000 tuples, distributed over networks of different sizes: 2,000; 4,000; 6,000; 8,000 and 10,000 nodes. The queries asked for a range covering 1% of the attribute domain.

From the results shown in Figure B.6, we see that the average cost of lookups increase linearly with network size. The great variations in number of messages are a result of the area that is covered by a query. The query area is a function of the width of the query and the central point. Queries that are close to the edges of the attribute domain cover a smaller area of the CAN space than do queries covering the central part of the domain.

The number of messages used for range index lookup increases more slowly than the number used by the baseline, so the distance from the baseline increases with network size.

Varying Query Range

Our next experiment shows how the other important parameter, the width of the query range, affects the number of messages. The varying query ranges should also have an effect on the precision of the queries. The experiment was done on a network

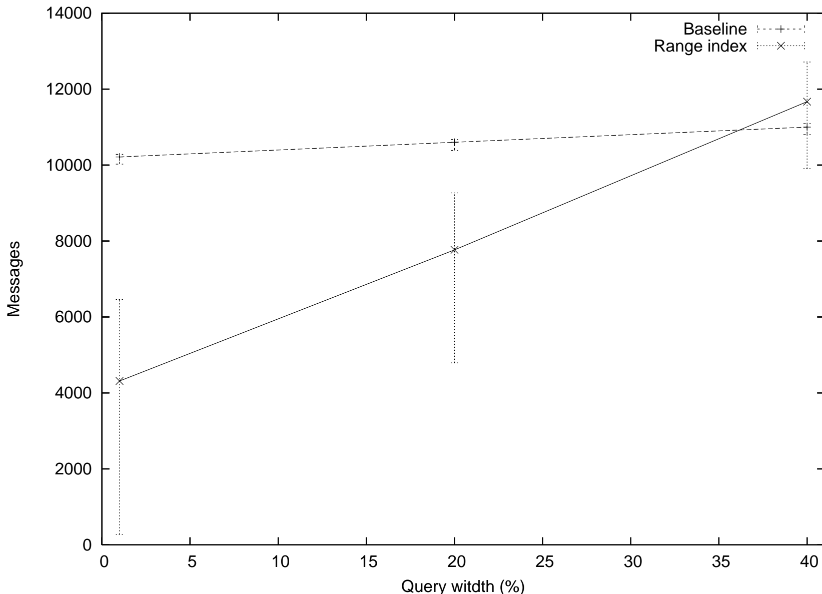


Figure B.7: Messages used when varying query widths.

of 10,000 nodes, storing a database of 10,000,000 tuples. The number of messages used was measured for queries covering 1%, 20% and 40% of the attribute domain.

The results are shown in Figure B.7. We see that the baseline method is nearly constant. The reason for this is that the only effect that increases the cost of this method is the number of ranges that are within the query range.

The cost of the range index method increases to more than the cost of the baseline method at a query width of about 37% of the domain width. After this point, the cost of locating nodes is higher than contacting all of them.

B.7 Conclusion

We have presented a method for distributed indexing of ranges instead of tuples in a peer-to-peer system. This indexing method significantly reduces the cost of inserts, updates and deletes, since index records do not have to be updated every time a tuple is changed. The range index can be used for data localization in a RDBMS.

Unlike previous peer-to-peer indexing methods, data placement is decoupled from the index structure. This allows for a greater degree of node autonomy and greater flexibility in data placement.

We also look at the cost of index lookups and show that index lookups are more efficient for narrow searches, but that it at some point becomes more efficient just to broadcast the query to all nodes, since so many of them are involved anyway.

As far as we know, this is the first peer-to-peer range index, and there are still

unsolved problems. Our index requires a specific kind of multidimensional DHT. Many systems use one-dimensional DHTs, and the method should be generalized to be used also in these systems.

More work should be done on answering queries by using statistics stored in index records. Summary queries, such as aggregation queries, will benefit from not having to check every tuple. Many systems require only approximate answers, and this would fit well in with the current indexing method, without requiring too frequent updates.

Bibliography

[1] M. Abdallah and H. C. Le. Scalable range query processing for large-scale distributed database applications. In *Proceedings of PDCS'2005*, 2005.

[2] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(3):29–33, 2003.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of SODA'2003*, 2003.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM'2004*, 2004.

[5] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of WebDB'04*, New York, NY, USA, 2004.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of KDD'1996*, 1996.

[7] J. Gao and P. Steenkiste. Efficient support for range queries in DHT-based systems. Technical Report CMU-CS-03-215, Carnegie Mellon University, 2003.

[8] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of CIDR'2003*, 2003.

[9] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: a scalable overlay network with practical locality properties. In *Proceedings of USENIX Symposium on Internet Technologies and Systems'2003*, 2003.

[10] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of SPAA'2002*, 2002.

[11] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *Proceedings of VLDB'2005*, 2005.

[12] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS'2002*, 2002.

[13] N. Ntarmos, T. Pitoura, and P. Triantafillou. Range query optimization leveraging peer heterogenity in DHT data networks. In *Proceedings of DBISP2P'2005*, 2005.

[14] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Proceedings of EDBT'2006*, 2006.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM'01*, 2001.

[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware'2001*, 2001.

[17] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Proceedings of P2P'2005*, 2005.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM'01*, 2001.

[19] P. Yalagandula and J. C. Browne. Solving range queries in a distributed system. Technical Report TR-04-18, Department of Computer Sciences, University of Texas at Austin, 2004.

[20] C. Zhang, A. Krishnamurthy, and R. Y. Wang. SkipIndex: towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton University Computer Science Department, 2004.

[21] M. Zhang and K.-L. Tan. Supporting rich queries in DHT-based peer-to-peer systems. In *Proceedings of WETICE'2003*. IEEE Computer Society, 2003.

[22] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *Proceedings of IPTPS'2006*, 2006.

[23] M. Zhou, R. Zhang, W. Qian, and A. Zhou. GChord: indexing for multi-attribute query in P2P system with low maintenance cost. In *Proceedings of DASFAA'2007*, 2007.

Paper C

Efficient and Robust Database Support for Data-Intensive Applications in Dynamic Environments

Jon Olav Hauglid, Kjetil Nørvåg and Norvald H. Ryeng.
In *Proceedings of ICDE*, 2009.

Abstract

Requirements from new types of applications call for new database system solutions. Computational science applications performing distributed computations on Grid networks with requirements for efficient storage and query solutions are now emerging. For this purpose we have developed DASCOSA-DB, a P2P-based distributed database system, which in addition to providing location-transparent storage and querying, also includes novel features like efficient partial restart of queries and redistribution of query operators in the context of failure, dynamic refragmentation and allocation, and distributed semantic caching. In this demo, the novel features will be demonstrated, combined with a more general description of the architecture and demonstration of the distributed query processing capabilities.

C.1 Introduction

Requirements from new types of applications call for new database system solutions. Computational science applications performing distributed computations on Grid networks with requirements for efficient storage and query solutions are now emerging.

While Grid *computing* has gained maturity through the recent years, management of data in Grid systems is less mature. Data storage and access is still mostly file oriented, and it is in general left to users to manage files and their locations as needed. Although some support has emerged for metadata management, more advanced database services is still only the proposal stage, examples are the OGSA-DAI and OGSA-DQP frameworks [2], which are service-based, with little cooperation between sites.

The goal of our research is a reliable *Database Grid*, with location-transparent storage, i.e., users/applications do not have to care about where data is stored and where queries are processed. The aim is sites cooperating on data storage and processing while retaining autonomy, i.e., a Grid-wide database system. A sub-goal (but maybe just as important in the long term!) is to help in making computational engineering society believe in databases!

What we provide is a SQL-accessible distributed database system supporting traditional features, but in addition also novel features intended to make the system more suitable for challenging dynamic environments:

- Automatic management of fragment location and replication.
- Efficient handling of query failures through a new partial restart technique.
- Distributed semantic caching.

In the reminder of the paper we will 1) present the background for our project, 2) give an overview of the architecture and implementation of DASCOSA-DB, with emphasis on three of the novel contributions offered by DASCOSA-DB, and 3) outline our three planned demonstrations.

C.2 Background

DASCOSA-DB can be considered somewhere in between a traditional distributed/federated database system and P2P DBMS. We will in this section describe some issues that forms the background for some of the design decisions in the development of DASCOSA-DB, as well as describing some related work.

Each DASCOSA-DB site has a large degree of local autonomy, but a high degree of cooperation (for example during query execution) is possible. Unlike a typical P2P setting, the participants in a Grid will have knowledge of each other, for example, which universities or companies are participating. Because of this, users can also be expected to be more "well-behaving" compared to a P2P network where most users can be assumed to only gain without giving if given the possibility. Note that although the participating organizations can be known, individual machines as such does not need to be known, i.e., each site will only know a few other sites.

It can be expected that in our application area, large and long-running queries involving many computers will be frequent. Since the probability of failure increases with query duration and number of computers involved, failure of individual sites (or connection to sites) during execution of a query can be frequent, and there is also a certain probability of double-failures, where also the restarted query fails. In some cases there can also be a deadline on data delivery, for example in combination with simulations/observation of physical processes. Thus, complete restart of a query should be avoided. It is also desirable that the handling of query failures should be completely transparent from applications. This contrasts to the traditional case where a query is aborted and has to be restarted.

A typical access pattern for many computational science applications is to first read an amount of initialization data, then sporadic queries and updates during computation, and finishing with writing a possibly large amount of result data to the database. The different sites will in general be in different phases wrt. access pattern. Global queries, for example from previous results, might be performed during execution. In many cases, strict transaction consistency and isolation is not critical. As a result, there is little need for optimizing on concurrency. However, since there can be large amounts of data created, this data should as much as possible be stored locally. In order to cope with this access pattern which is dynamic both with respect to site and data, dynamic fragmentation and replication is needed.

During the recent years, a large number of research papers on topics related to indexing and querying structured data in P2P network has been published. Of particular interest in the context of our work are a number of other projects aiming at providing *relational* data access through P2P technology like PIER [6], PeerDB [7], Hyperion [10], and APPA [1]. Also other distributed storage systems like, e.g., Bigtable [3] provide distributed storage of data but no query capability. We will in this paper focus on features in DASCOSA-DB not available in other systems.

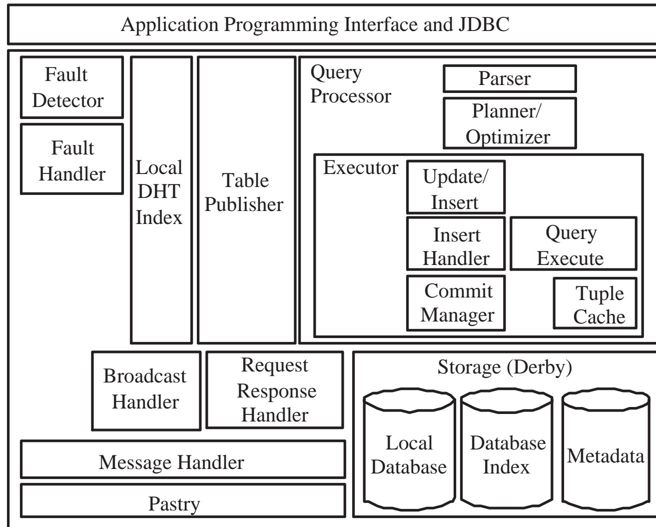


Figure C.1: High-level overview of the architecture of DASCOSA-DB.

C.3 Overview of DASCOSA-DB

In this section we give an overview of 1) the architecture of DASCOSA-DB with particular emphasis on the its novel partial restart techniques, adaptive fragment management and distributed semantic caching, and 2) the implementation of the current version of DASCOSA-DB. For more information about details of DASCOSA-DB we refer to the project's web page.[1]

C.3.1 Architecture

The global data model used in DASCOSA-DB is based on the relational model. A table can be stored in its entirety on one site, or it can be horizontally fragmented over a number of sites. In order to improve both performance as well as availability, fragments can be replicated, i.e., fragments of a table can be stored on more than one site.

In some approaches to P2P databases, every tuple is individually indexed in the P2P system (typically using a DHT *put* operation). In the case of large amounts of data and updates, and with queries more complex than single lookup operations, such solutions are not scalable. In DASCOSA-DB, we instead use the DHT to index data of larger granularity (fragments), i.e., the DHT can be considered as a highly reliable and distributed catalog. The main task of the DHT-part of the system is to provide a mapping between a table name and the sites storing the fragments of the table. Information about local fragments is regularly published to the DHT.

[1]`http://research.idi.ntnu.no/dascosa/`

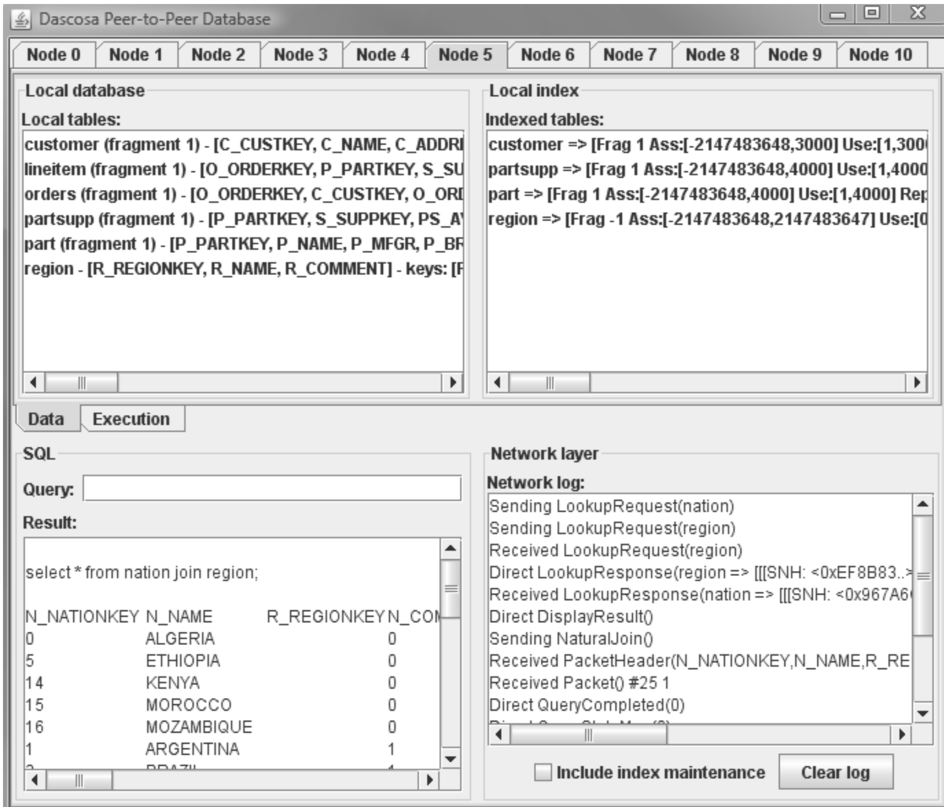


Figure C.2: Screenshot from the DASCOSA-DB system monitoring tool.

The overall architecture of DASCOSA-DB is illustrated in Fig. C.1. As can be seen from the figure, DASCOSA-DB can be viewed as middleware on top of a DHT and local database system.

Query processing is in DASCOSA-DB performed quite similarly to traditional databases, where SQL is transformed into a distributed execution plan (i.e., query operators annotated with the site where they are going to be executed). After planning, query execution begins by transmitting the algebra tree from the initiator site to the different sites involved. Choice of sites is based on location of fragments and total reduction of communication cost.

C.3.2 Novel Features

We now give a brief description of 3 novel features of DASCOSA-DB: 1) partial query restart, 2) dynamic fragment management, and 3) distributed semantic caching.

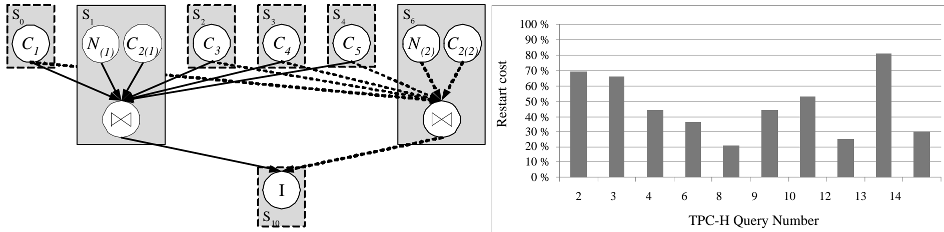


Figure C.3: Query and performance under churn. Replica j of fragment T_i is denoted $T_{i(j)}$.

Partial Restart

As mentioned above, the probability of failure during a query increases with the number of sites involved in the query and with longer duration of queries and/or higher churn rate (unavailable sites). Traditionally, only failure during update transactions has been considered, and failure of queries has been handled by complete query restart. While this is an appropriate technique for small and medium-sized queries, it can be expensive for very large queries, and in some application areas there can also be deadlines on results so that complete restart should be avoided. While in some cases various checkpoint-restart techniques have been employed to avoid complete restart of operations, these techniques have been geared towards update/load operations, and in many cases implies that a query will be delayed until the failed site is online again.

An alternative to local checkpointing and complete restart is a technique supporting *partial restart*. In this case, unfinished subqueries from failed sites can be resumed on new sites after failures, and utilizing partial results already produced before the failure (both results generated at non-failing sites as well as results from failing sites that have already been communicated to non-failing sites). In DASCOSA-DB we have integrated a new technique for partial restart that compared to previous approaches like [11] 1) reduces query execution time compared to complete restart, 2) incurs minimal extra network traffic during recovery from query failure, 3) employs decentralized failure detection, 4) supports non-blocking operators, 5) handles recovery from multi-site failures, and 6) avoids duplicate tuples by deterministic delivery of tuples from base relations and operators..

An example of partial restart is illustrated in Fig. C.3. To the left is illustrated 6 sites, each storing fragments of the *customer* and *nation* tables of the TPC-H benchmark (all sites store fragments of *customer*, there is a replica of fragment 2 on both site S_1 and S_5, and the single fragment of *nation* is stored on both site S_1 and S_5). Assume then that the simple query
`SELECT * FROM nation JOIN customer`
is issued from site S_{10}, resulting in the query tree in Fig. C.3 (site S_1 was selected for the join operator during planning to minimize network traffic as it has a fragment of both involved tables). Then assume that site S_1 fails sometime during the

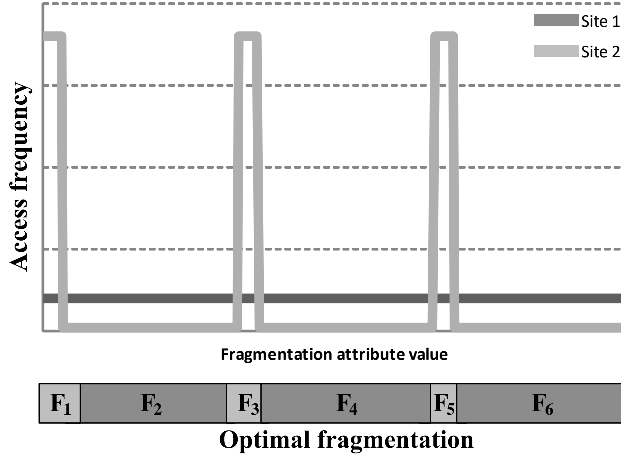


Figure C.4: Example access pattern, and desired fragmentation and allocation.

query. The failure is discovered by site S_{10} which select site S_5 as replacement site. The particular challenges that have been solved in our approach relates to failure detection, selection of replacement site, and restart of the various relational algebra operators. In Fig. C.3 is illustrated the restart cost for some selected TPC-H queries, compare to complete restart. As can be shown, the cost of restart is considerably reduced using partial restart.

Dynamic Fragment Management

Traditionally, table fragments in distributed database systems have been fragmented and replicated based on fixed value ranges or rules defined by database administrators. In DASCOSA-DB, fragments and replicas are created and migrated automatically by the system based on the access pattern, aiming at keeping the amount of accesses to remote sites as low as possible. The ranges of fragments are not fixed, so that fragments can be split and coalesced automatically. In this way, the system will be able to efficiently adapt to changing workloads.

An example of what we aim at with our approach is illustrated in Fig. C.4, where the figure illustrates the access pattern to a database table from two sites. Site 1 has a uniform distribution of accesses, while Site 2 has an access pattern with distinct hot spots. In this case, a good fragmentation would be 6 fragments, one for each of the hot spot areas and one for each of the areas between. A good allocation would be the fragments of the hot spot areas (F_1, F_3, and F_5) allocated to site 2, with the other fragments (F_2, F_4, and F_6) allocated to site 1. Using our approach this access pattern will be detected, and fragmentation and allocation performed accordingly. Note that if the access pattern changes later, this will be detected and fragments reallocated in addition to possible repartitioning.

In DASCOSA-DB, access statistics is used to detect pattern. For each fragment,

a set of histograms is maintained, and each histogram represents statistics about accesses from one particular remote site. Each bucket in a histogram represents a value subrange of the fragment, and contains an estimate of the number of accesses to the actual interval the bucket covers. At regular times, the histograms are analyzed, and it is determined through the use of cost functions whether the overall cost would be lower if a fragment were located on another site. It can also be detected if a subinterval of a fragment is heavily accessed from a remote site. In this case, it can be decided that the fragment should be split into 3 fragments, and the fragment containing the interval heavily accessed from the remote site is migrated to the remote site. In order to avoid too many fragments, fragments which meets in the value interval can be coalesced when they end up at the same site. I.e., a fragment resulted from a split and then migrated might actually be coalesced with another fragment when it is received at the remote site (whether this is performed or not is also based on considerations of potential replicas of the fragments). Contents in histograms are regularly expired, so that they only contain recent statistics and only represent remote sites that have recently accessed the fragment.

It is important to note that our approach for fragment maintenance is different from previous approaches (e.g., [9]) that are based on analyzing SQL queries, while in our approach accesses at tuple level are considered. Also the possibility of managing higher degree of dynamics as well as migration strategies distinguish our techniques from previous approaches.

Distributed Semantic Caching

In semantic caching, results from selected queries as well as their query descriptions are kept. These results can be applied to reuse results from previous queries, thus reducing the total query cost. In large-scale distributed database systems, using a central central server with complete knowledge of the system will be a serious bottleneck and single point of failure. In DASCOSA-DB this problem is reduced by distributing the caching knowledge over several sites. In addition, decisions about what to cache and cache replacement is performed automatic and site-autonomous.

C.3.3 Implementation

The implementation of DASCOSA-DB has been performed in two steps: first, a simple proof-of-concept prototype was developed [8]. The current version is a completely re-implemented version, and acts as a middleware on top of Apache Derby [4] running as the local DBMS on each site. The catalog service for indexing tables is implemented using the FreePastry DHT [5]. DASCOSA-DB is 100% Java, which gives easy deployment, platform independence, and can also be embedded in other software if desired.

In order to facilitate easy interactive access to the system as well as study configuration, distribution of data, and query execution, we have implemented a monitoring tool, cf. the screenshot in Fig. C.2. As can be seen in the figure, queries can be entered and information about local tables as well as information about remote tables

(which is part of the responsibility of the site as participant in the DHT) can be found. Both static statistics as well as per-query statistics can easily be viewed.

C.4 Demonstration

Our demonstration will illustrate both the general distributed database aspects of DASCOSA-DB, as well as more novel aspects like partial restart. In the demonstration we will for convenience run a number of DASCOSA-DB-instances on one or two machines. The demonstration will use a dataset generated by the TPC-H data generator [12].

C.4.1 Overall and Distributed Queries

In this demonstration we will give an overview of DASCOSA and its basic features. We will show how both simple and complex SQL queries are executed in the system, and demonstrate how fragmentation and replication aspects are automatically handled by the system.

C.4.2 Partial Restart

In this demonstration we will demonstrate how DASCOSA-DB work in the context of failing sites during query execution. The effect of partial restart will be demonstrated by queries not failing, queries failing and not employing partial restart (i.e., complete restart), and employing our partial restart approach. We will in this context also show how the system automatically selects new sites that will complete the work of failed sites, and how replication can make restart possible even when sites storing base tables fail during a query.

C.4.3 Distributed Semantic Caching

In this demonstration we will show how DASCOSA-DB utilizes cached subqueries in order to improve performance in the context of repeated queries or queries that contains subqueries of previous queries.

C.5 Future Work

Although we now have a working prototype of a distributed database system, there is no lack of remaining challenges. Improving query optimization in the context of adaptive fragments and replication is an obvious goal. Another important issue in our context is more automatic handling schema heterogeneity, where the plan is to employ ontology-based methods. We also intend to study how equivalents of partial restart and adaptive fragmentation can be employed in the context of XML data.

Acknowledgments

The authors would like to thank other previous and current participants in the DASCOSA project: Eirik Eide, Odin H. Standal, and João Rocha. Supported by grant #176894/V30 from the Norwegian Research Council.

Bibliography

[1] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Design and implementation of Atlas P2P architecture. In *Global Data Management*, 2006.

[2] M. N. Alpdemir et al. OGSA-DQP: a service for distributed querying on the Grid. In *Proceedings of EDBT'2004*, 2004.

[3] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Proceedings of OSDI'2006*, 2006.

[4] Apache Derby, `http://db.apache.org/derby/`.

[5] FreePastry, `http://freepastry.org/`.

[6] R. Huebsch et al. Querying the internet with PIER. In *Proceedings of VLDB'2003*, 2003.

[7] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of ICDE'2003*, 2003.

[8] K. Nørvåg. DASCOSA: database support for computational science applications. In *Proceedings of GLOBE'06*, 2006.

[9] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *Proceedings of SIGMOD'2002*, 2002.

[10] P. Rodríguez-Gianolli et al. Data sharing in the Hyperion peer database system. In *Proceedings of VLDB'2005*, 2005.

[11] J. Smith and P. Watson. Fault-tolerance in distributed query processing. In *Proceedings of IDEAS'2005*, 2005.

[12] TPC-H, `http://www.tpc.org/tpch/`.

Paper D

DYFRAM: Dynamic Fragmentation and Replica Management in Distributed Database Systems

Jon Olav Hauglid, Norvald H. Ryeng and Kjetil Nørvåg.
In *Distributed and Parallel Databases* 28(2–3), pages 157–185, 2010.

Abstract

In distributed database systems, tables are frequently fragmented and replicated over a number of sites in order to reduce network communication costs. How to fragment, when to replicate and how to allocate the fragments to the sites are challenging problems that has previously been solved either by static fragmentation, replication and allocation, or based on a priori query analysis. Many emerging applications of distributed database systems generate very dynamic workloads with frequent changes in access patterns from different sites. In such contexts, continuous refragmentation and reallocation can significantly improve performance. In this paper we present DYFRAM, a decentralized approach for dynamic table fragmentation and allocation in distributed database systems based on observation of the access patterns of sites to tables. The approach performs fragmentation, replication, and reallocation based on recent access history, aiming at maximizing the number of local accesses compared to accesses from remote sites. We show through simulations and experiments on the DASCOSA distributed database system that the approach significantly reduces communication costs for typical access patterns, thus demonstrating the feasibility of our approach.

D.1 Introduction

There is an emerging need for efficient support of databases consisting of very large amounts of data that are created and used by applications at different physical locations. Examples of application areas include telecom databases, scientific databases on grids, distributed data warehouses, and large distributed enterprise databases. In many of these application areas the delay from accessing a remote database is still significant enough to make necessary the use of distributed databases employing fragmentation and replication, a fact also evident recently by increased support for distributed fragmented and replicated tables in commercial products like MySQL Cluster.

In distributed databases, the communication costs can be reduced by partitioning database tables horizontally into *fragments*, and allocating these fragments to the sites where they are most frequently accessed. The aim is to make most data accesses local, and avoid remote reads and writes. The read cost can be further reduced by the replication of fragments when beneficial. Obviously, important challenges in fragmentation and replication are *how to fragment*, *when to replicate fragments*, and *how to allocate the (replicated) fragments*.

Previous work on data allocation has focused on (mostly static) fragmentation based on analyzing queries. These techniques are only useful in contexts where read queries dominate and where decisions can be made based on SQL-statement analysis. Moreover, they also involve centralized computations based on collected statistics from participating sites. However, in many application areas, workloads are very dynamic with frequent changes in access patterns at different sites. One common reason for this is that their data usage often consists of two separate phases: a first phase where writing of data dominates (for instance during simulation when

results are written), and a subsequent second phase when a subset of the data, for example results, is mostly read. The dynamism of the overall access pattern is further increased by different instances of the applications executing in different phases at different sites.

Because of dynamic workloads, static/manual fragmentation and replication may not always be optimal. Instead, the fragment and replication management should be dynamic and completely automatic, i.e., changing access patterns should result in refragmentation and reallocation of fragments when beneficial, as well as in the creation or removal of fragment replicas. In this paper, we present *DYFRAM,* a decentralized approach for dynamic fragmentation and replica management in distributed database systems, based on observation of access patterns of sites to tables. Fragmentation and replication is performed based on recent access history, aiming at maximizing the number of local accesses compared to accesses from remote sites.

An example of what we aim at achieving with our approach is illustrated in Fig. D.1. It illustrates the access pattern of a database table from two sites. Site 1 has a uniform distribution of accesses, while site 2 has an access pattern with distinct hot spots. In this case, a good fragmentation would generate 6 fragments, one for each of the hot spot areas and one for each of the intermediate areas. A good allocation would be the fragments of the hot spot areas (F_1, F_3, and F_5) allocated to site 2, with the other fragments (F_2, F_4, and F_6) allocated to site 1. As will be shown later in the experimental evaluation, DYFRAM will detect this pattern, split the table into appropriate fragments, and then allocate these fragments to the appropriate sites. Whether some of the fragments should be replicated or not depends on the read/write pattern. Note that if the access pattern changes later, this will be detected and fragments reallocated as well as repartitioned if necessary.

The main contributions of this paper are 1) a low-cost algorithm for fragmentation decisions, making it possible to perform refragmentation based on the recent workload, and 2) dynamic reallocation and replication of fragments in order to minimize total access cost in the system. The process is performed in a completely decentralized manner, i.e., without a particular controlling or coordinating site. An important aspect of our approach is *the combination of the dynamic refragmentation, reallocation, and replication into a unified process.* To the best of our knowledge, no previous work exists that perform this task dynamically during query execution based on both reads and writes in a distributed setting. Our approach is also applicable in a parallel system, since one of our important contributions compared to previous work is that the decisions can be taken without communication of statistics or synchronization between sites.

The organization of the rest of this paper is as follows. In Section D.2 we give an overview of related work. In Section D.3 we outline our system and fragment model and state the problem tackled in this work. In Section D.4 we give an overview of DYFRAM. In Section D.5 we describe how to manage replica access statistics. In Section D.6 we describe in detail the dynamic fragmentation and replication algorithm. In Section D.7 we evaluate the usefulness of our approach. Finally, in Section D.8, we conclude the paper and outline issues for further work.

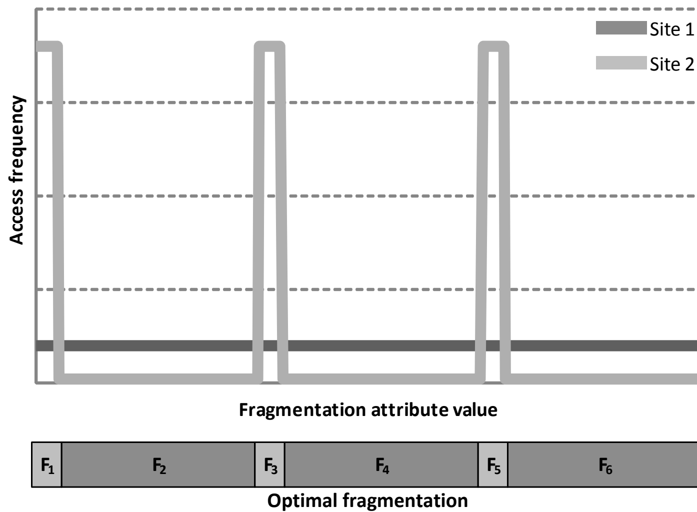


Figure D.1: Example access pattern, and desired fragmentation and allocation.

D.2 Related Work

The problem of fragmenting tables so that data is accessed locally has been studied before. It is also related to some of the research in distributed file systems (see a summary in [14]). One important difference between distributed file systems and distributed database systems is the typical granularity of data under consideration (files vs. tuples) and the need for a fragmentation attribute that can be used for partitioning in distributed database systems.

Fragmentation is tightly coupled with fragment allocation. There are methods that do only fragmentation [2, 24, 26, 33, 34] and methods that do only allocation of predefined fragments [3, 4, 7, 10, 13, 20, 30]. Some methods also exist that integrate both tasks [9, 11, 17, 19, 25, 27, 29]. Replication, however, is typically done as a separate task [5, 8, 15, 21, 22, 32], although some methods, like ours, take an integral view of fragmentation, allocation and replication [11, 27, 29]. Dynamic replication algorithms [5, 15, 21, 22, 32] can optimize for different measures, but we believe that refragmentation and reallocation must be considered as alternatives to replication. In DYFRAM we choose among all these options when optimizing for communication costs. Our replication scheme is somewhat similar to that of DIBAS [11], but DYFRAM also allows remote reads and writes to the master replica, whereas DIBAS always uses replication for reads and do not allow remote writes to the master replica. This operation shipping is important when analyses [8] of replication vs. remote reads and writes conclude that the replication costs in some cases may be higher than the gain from local data access. A key difference between DIBAS and DYFRAM is that DIBAS is a static method where replication is based on offline analysis of database accesses, while DYFRAM is dynamic and does replication

online as the workload changes.

Another important categorization of fragmentation, allocation and replication methods is whether they are static or dynamic. Static methods analyze and optimize for an expected database workload. This workload is typically a set of database queries gathered from the live system, but it could also include inserts and updates. Some methods also use more particular information on the data in addition to the query set [26]. This information has to be provided by the user, and is not available in a fully automated system. A form of static method is the design advisor [34] which suggests possible actions to a database administrator. The static methods are used at major database reconfigurations. Some approaches, such as evolutionary algorithms for fragment allocation [3, 10], lend themselves easily to the static setting.

Static methods look at a set of queries or operations. It can be argued that the workload should be viewed as a sequence of operations, not as a set [1]. Dynamic methods continuously monitor the database and adapt to the workload as it is at the moment and are thus viewing a sequence of operations. Dynamic methods are part of the trend towards fully automatic tuning [31], which has become a popular research direction. Recently, work has appeared aiming at integrating vertical and horizontal partitioning while also taking other physical design features like indices and materialized views into consideration [2]. Adaptive indexing [1, 6] aims to create indices dynamically when the costs can be amortized over a long sequence of read operations, and to drop them if there is a long sequence of write operations that would suffer from having to update both base tables and indices. Our work is on tables and table fragments, but shares the idea of amortizing costs over the expected sequence of operations. In adaptive data placement, the focus has either been on load balancing by data balancing [9, 17], or on query analysis [19]. In our algorithms, we seek to place data on the sites where they are being used (by reads or writes), not to balance the load.

Using our method, fragments are automatically split, coalesced, reallocated and replicated to fit the current workload using fragment access statistics as a basis for fragment adjustment decisions. When the workload changes, our method adjusts quickly to the new situation, without waiting for human intervention or major reconfiguration moments. Closest to our approach may be the work of Brunstrom et al. [7], which studied dynamic data allocation in a system with changing workloads. Their approach is based on pre-defined fragments that are periodically considered for reallocation based on the number of accesses to each fragment. In our work, there are no pre-defined fragments. In addition to reallocating, fragments can be split and coalesced on the fly. Our system constantly monitors access statistics to quickly respond to emerging trends and patterns.

A third aspect is how the methods deal with distribution. The method can either be centralized, which means that a central site gathers information and decides on the fragmentation, allocation or replication, or it can be decentralized, delegating the decisions to each site. Some methods use a weak form of decentralization where sites are organized in groups, and each group chooses a coordinator site that is charged with making decisions for the whole group [15, 21].

Among the decentralized systems, we find replication schemes for mobile ad hoc

networks (see [23] for an overview). However, these approaches do not consider table fragmentation and in general do replication decisions on a more coarse granularity, e.g., files.

In DYFRAM, fragmentation, allocation and replication decisions are fully decentralized. Each site decides over its own fragments, and decisions are made on the fly based on current operations and recent history of local reads and writes. Contrary to much of the work on parallel database systems, our approach has each site as an entry point for operations. This means that no single site has the full overview of the workload. Instead of connecting to the query processor and reading the WHERE-part of queries, we rely on local access statistics.

Mariposa [27, 28] is a notable exception to the traditional, manually fragmented systems. It provides refragmentation, reallocation and replication based on a bidding protocol. The difference from our work is chiefly in the decision-making process. A Mariposa site will sell its data to the highest bidder in a bidding process where sites may buy data to execute queries locally or pay less to access it remotely with larger access times, optimizing for queries that have the budget to buy the most data. A DYFRAM site will split off, reallocate or replicate a fragment if it optimizes access to this fragment, seen from the fragment's viewpoint. This is performed also during query execution, not only as part of query planning, as is the case in Mariposa.

A summary and feature comparison of our method and related fragmentation, allocation and replication methods is given in Table D.1. We show which features are provided by each method and whether it is a dynamic method that adapts to the workload or a static method that never updates its decision. The methods are also categorized according to the where the decisions to fragment, allocate and replicate are made. This can be done either centralized to a single site which has the necessary information about the other sites, or decentralized.

D.3 Preliminaries

In this section we provide the context for the rest of the paper. We introduce symbols to be used throughout the paper, which are shown in Table D.2.

D.3.1 System Model

The system is assumed to consist of a number of sites $S_i, i = 1 \dots n$, and we assume that sites have equal computing capabilities and communication capacities. Each site runs a DBMS, and a site can access local data and take part in the execution of distributed queries, i.e., the local DBMSs together constitute a distributed database system. The distribution aspects can be supported directly by the local DBMS or can be provided through middleware.

Metadata management, including information on fragmentation and where replicas are stored, is performed through a common catalog service. This catalog service can be realized in a number of ways, for example in our prototype system we use a distributed hash table where all sites participate [16].

Table D.1: Summary of related fragmentation, allocation and replication methods.

	Fragmentation	Allocation	Replication	Dynamic	Static	Centralized	Decentralized
DYFRAM	✓	✓	✓	✓			✓
Agrawal et al. [2]	✓				✓	✓	
Ahmad et al. [3]		✓			✓	✓	
Apers [4]		✓			✓	✓	
Bonvin et al. [5]			✓	✓			✓
Brunstrom et al. [7]		✓		✓			✓
Ciciani et al. [8]			✓		✓	✓	
Copeland et al. [9]	✓	✓		✓		✓	
Corcoran and Hale [10]		✓			✓	✓	
Didriksen and Galindo-Legaria [11]	✓	✓	✓		✓	✓	
Furtado [13]		✓			✓	✓	
Hara and Madria [15]			✓	✓			✓
Hua and Lee [17]	✓	✓		✓		✓	
Ivanova et al. [19]	✓	✓		✓		✓	
Menon [20]		✓			✓	✓	
Mondal et al. [21]			✓	✓			✓
Mondal et al. [22]			✓	✓			✓
Rao et al. [24]	✓				✓	✓	
Saccà and Wiederhold [25]	✓	✓			✓	✓	
Shin and Irani [26]	✓				✓	✓	
Sidell et al. [27]	✓	✓	✓	✓			✓
Tamhankar and Ram [29]	✓	✓	✓		✓	✓	
Ulus and Uysal [30]		✓		✓			✓
Wolfson and Jajodia [32]			✓	✓			✓
Wong and Katz [33]	✓				✓	✓	
Zilio et al. [34]	✓				✓	✓	

Our approach assumes that data can be represented in the (object-)relational data model, i.e., tuples t_i being part of a table T. A table can be stored in its entirety on one site, or it can be horizontally fragmented over a number of sites. Fragment i of table T is denoted F_i.

In order to improve performance as well as availability, fragments can be replicated, i.e., a fragment can be stored on more than one site. We require that replication is master-copy based, i.e., all updates to a fragment are performed to the master-copy, and afterward propagated to the replicas. If a master replica gets refragmented, other replicas must be notified so they can be refragmented as well.

D.3.2 Fragment Model

Fragmentation is based on one attribute value having a domain D, and each fragment covering an interval of the domain of the attribute, which we call *fragment value domain* (FVD). We denote the fragment value domain for a fragment F_i as $FVD(F_i) = F_i[min_i, max_i]$. Note that the FVD does not imply anything about what values that actually exist in a fragment. It only states that if there is a tuple in the global table with value v in the fragmentation attribute, then this tuple will be in the fragment with the FVD that covers v. We define two fragments F_i and F_j to be *adjacent* if their FVD meets, i.e.:

$$adj(F_i, F_j) \Rightarrow max_i = min_j \vee max_j = min_i$$

When a table is first created, it consists of one fragment covering the whole domain of the fragmentation attribute value, i.e., $F_0[D_{min}, D_{max}]$, or the table consists of a number of fragments $F_1, \ldots, F_n$ where $\cup_{i=1}^{n} FVD(F_i) = [D_{min}, D_{max}]$. A fragment F_{old} can subsequently be split into two or more fragments $F_1, \ldots, F_n$. In this case, the following holds true:

$$\cup_{i=1}^{n} F_i = F_{old}$$

$$\forall F_i, F_j \in \{F_1, \ldots, F_n\} F_i \neq F_j \Rightarrow F_i \cap F_j = \emptyset$$

In other words, the new fragments together cover the same FVD as the original fragment, and they are non-overlapping. Two or more adjacent fragments $F_1, \ldots, F_n$ can also be coalesced into a new fragment if the new fragment covers the same FVD as the previous fragments covered together:

$$F_{new} = \cup_{i=1}^{n} F_i$$

$$\forall F_i \in \{F_1, \ldots, F_n\}, \exists (F_j \in \{F_1, \ldots, F_n\}) : adj(F_i, F_j)$$

Consider a distributed database system consisting of a number of sites $S_i, i = 1 \ldots n$ and a global table T. At any time the table T has a certain fragmentation, e.g., $\mathcal{F} = \{S_0(F_0, F_3), S_3(F_1, F_2)\}$. Note that not all sites have been allocated fragments, and that there might be replicas of fragments created based on the read pattern. In this case, we distinguish between the master replica R^m where the updates will be applied, and the read replicas R_i^r. Using a master-copy protocol the read replicas R_i^r will receive updates after they have been applied to the master replica R^m.

Table D.2: Symbols.

Symbol	Description
S_i	Site
t_i	Tuple
T	Table T
F_i	Fragment i of table T
R_i	Replica i
R^m	Master replica
$F_i[min, max]$	Fragment value domain
$\mathcal{F}$	Fragmentation
C	Cost
A_i	Tuple access
RE_j	Refragmentation

D.3.3 Problem Definition

During operation, tuples are accessed as part of read or write operations A. If the fragment where a tuple belongs (based on the value of the fragmentation attribute) is stored on the same site as the site S_a performing the read access A_R, it is a local read access and the cost is $C(A_R) = C_L$. On the other hand, if the fragment is stored on a remote site, a remote read access has to be performed, which has a cost of $C(A_R) = C_R$.

In the case of a write access, the cost also depends on whether the fragment to which the tuple belongs is replicated or not. The basic write cost of a tuple belonging to a master replica that is stored on the same site as the site S_a performing the write access is $C(A_W) = C_L$. If the master replica is stored on a remote site, a remote write access has to be performed, which has a cost of $C(A_W) = C_W$. In addition, if the fragment is replicated, the write will incur updates to the read replicas, i.e., $C(A_U) = rC_W$ where r is the number of read replicas.

In this paper we focus on reducing the communication costs, and therefore assume that $C_L = 0$. Note, however, that it is trivial to extend our approach by including local processing cost.

If we consider the accesses in the system as a sequence of n operations at discrete time instants, the result is a sequence of accesses $[A_1, ..., A_n]$. The total access cost is $\sum_i C(A_i)$. The access cost of a tuple at a particular time instant depends on the fragmentation $\mathcal{F}$.

Refragmentation and reallocation of replicas of fragments can be performed at any time. Given a computationally cheap algorithm for determining fragmentation and allocation, the main cost of refragmentation and reallocation is the migration or copying of fragments from one site to another. We denote the cost of one refragmentation or reallocation as $C(RE_j)$ (this includes any regeneration of indices after migration), and the cost of all refragmentations and reallocations as $\sum_j C(RE_j)$.

The combined cost of access, refragmentations and reallocations is thus $C_{total} =$

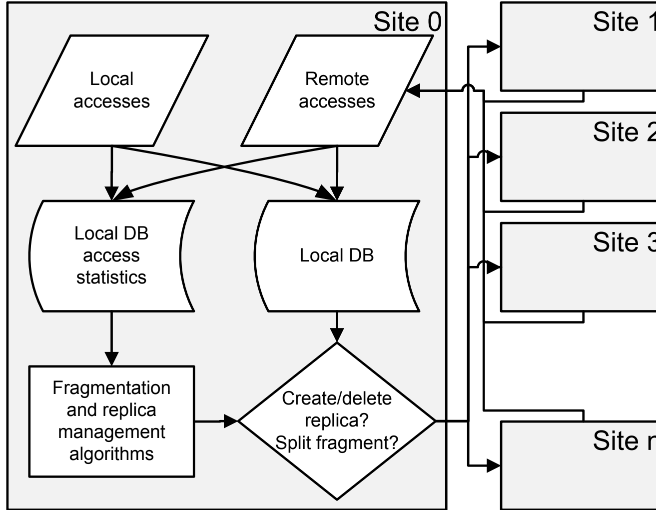


Figure D.2: Dynamic fragmentation and allocation.

$\sum_i C(A_i) + \sum_j C(RE_j)$. Note that the access, refragmentation and reallocation operations are interleaved. The aim of our approach is to minimize the cost C_{total}.

D.4 Overview of DYFRAM

This section describes our approach to dynamically fragment tables, and replicate those fragments on different sites in order to improve locality of table accesses and thus reduce communication costs. Our approach has two main components: 1) detecting replica access patterns, and based on these statistics to 2) decide on refragmentation and reallocation. The approach is illustrated in Fig. D.2.

Each site makes decisions to split, migrate and/or replicate independently of other sites. This makes it possible to use our approach without communication overhead, changing the network protocol or even using it on all sites in the system.

In order to make informed decisions about useful fragmentation and replica changes, future accesses have to be predicted. As with most online algorithms, predicting the future is based on knowledge of the past. In our approach, this means detecting replica access patterns, i.e., which sites are accessing which parts of which replica. This is performed by recording replica accesses in order to discover access patterns. Recording of accesses is performed continuously. Old data is periodically discarded so that statistics only include recent accesses. In this way, the system can adapt to changes in access patterns. Statistics are stored using histograms, as described in Section D.5.

Given the available statistics, our algorithm examines accesses for each replica and evaluates possible refragmentations and reallocations based on recent history. The algorithm runs at given intervals, individually for each replica. Since decisions are made independently of other sites, decisions are made based on the information

available at that site. With master-copy based replication, all writes are made to the master replica before read replicas are updated. Therefore, write statistics are available at all sites with a replica of a given fragment. On the other hand, reads are only logged at the site where the accessed replica is located. This means that read statistics are spread throughout the system. In order to detect if a specific site has a read pattern that indicates that it should be given a replica, we require a site to read from a specific replica so that this site's read pattern is not distributed among several replicas.

With all sites with replicas of a given fragment acting independently, we have to make sure that decisions taken are not in conflict with each other. To achieve this, we handle the master replica and read replicas differently. The site with the master replica can: 1) split the fragment, 2) transfer the master status to a different replica, and 3) create a new replica. Sites with read replicas can: 1) create a new replica, and 2) delete its own replica.

These decisions are made by the algorithm by using cost functions that estimate the difference in future communication costs between a given replica change and keeping it as is. Details are presented in Section D.6.

Regarding data consistency and concurrency control, this can be treated as in existing systems employing fragmentation and replication and is therefore not outlined here. In our DASCOSA-DB distributed database system [16], locking in combination with the system catalog (DHT-based) is used, however more complex protocols can also be used in order to increase concurrency (this is not specific to DYFRAM).

D.5 Replica Access Statistics

Recording of replica accesses is performed at the tuple level. The access data consists of (S, v, a) tuples, where S is the site from which the operation came, v is the value of the fragmentation attribute and a is the access type (read or write). In cases where recording every access can be costly (the overhead is discussed later), it is possible to instead record a sample of accesses — trading accuracy for reduced overhead.

The data structure used to store access statistics is of great importance to our approach. It should have the following properties:

- Must hold enough information to capture read and write patterns.
- Efficient handling of updates as they will be frequent.
- Memory efficient - storage requirements should not depend on fragment size or number of accesses.
- Must be able to handle any v values, because it will not be known beforehand which ranges are actually used.
- Must be able to effortlessly remove old access history in order to only keep recent history.

Table D.3: Histogram symbols.

Symbol	Description
H_i	Histogram
b_k	Histogram bucket number
$R_i[b_k]$	Number of reads in bucket
$W_i[b_k]$	Number of writes in bucket
W	Bucket width
MAX_B	Maximum number of buckets
Z_W	Factor used when resizing buckets

Since our purpose for recording accesses is to detect access patterns in order to support fragmentation decisions, we are interested in knowing how much any given site has accessed different parts of the fragment. We store access statistics in histograms. Every site has a set of histograms for each fragment it has a local replica of. These histograms must be small enough to be kept in main memory for efficient processing.

In the following, we present the design of our access statistics histograms as well as algorithms for the different histogram operations.

D.5.1 Histogram Design

Histograms have been used for a long time to approximate data distribution in databases [18]. Most of these have been *static histograms* constructed once and then left unchanged. In our case, data to be represented by the histograms arrive continuously. Static histograms would therefore soon be out of date and constant construction of new histograms would have prohibitive cost.

Another class of histograms is *dynamic histograms* [12, 18], that are maintained incrementally and therefore better suited for our approach. Most histograms described in the literature are equi-depth histograms, since these capture distributions better than equi-width histograms for the same number of buckets [18].

For our approach we chose to use equi-width histograms. This choice was made in order to improve the performance of histogram operations, since equi-width histograms are by design simpler to use and to access than equi-depth histograms. This is because all buckets have the same width, and finding the correct bucket for a given value is therefore a very simple computation. As will become apparent when we describe histogram updates and retrievals in detail below, it also simplifies computing histogram range counts when we use two different histogram sets in order to store only the recent history. The obvious disadvantage of using equi-width histograms is that we have to use more buckets in order to capture access patterns with the same accuracy as equi-depth histograms. However, the significantly reduced computational cost makes this an acceptable trade-off.

Histogram-related symbols used in the following discussion are summarized in Table D.3. Each bucket in a histogram H_i has a bucket number b_k and contains

two values: the read count $R_i[b_k]$ and the write count $W_i[b_k]$. We use equi-width histograms with bucket width W and limit bucket value ranges to start and end on multiples of W. The value range of a bucket is then $[b_k \cdot W, (b_k + 1) \cdot W)$.

Histograms only maintain statistics for values that are actually accessed, i.e., they do not cover the whole *FVD*. This saves space by not storing empty buckets, which is useful since we lack a priori knowledge about fragment attribute values. Buckets are therefore stored as $(b_k, R_i[b_k], W_i[b_k])$ triplets hashed on b_k for fast access.

In order to limit memory usage, there is a maximum number of stored buckets, MAX_B. If a histogram update brings the number of stored buckets above MAX_B, the bucket width is scaled up by a factor Z_W. Similarly, bucket width is decreased by the same factor if it can be done without resulting in more than MAX_B buckets. This makes sure we have as many buckets as possible given memory limitations, as this better captures the replica access history.

In order to store only the most recent history, we use two sets of histograms: the old and the current set. All operations are recorded in the current set. Every time the evaluation algorithms have been run, the old set is cleared and the sets swapped. This means that the current set holds operations recorded since the last time the algorithm was run, while the old set holds operations recorded between the two last runs. For calculations, the algorithms uses both sets. This is made simple by the fact that we always use the same bucket width for both sets and that bucket value range is a function of bucket number and width. Adding histograms is therefore performed by adding corresponding bucket values. We denote the current histogram storing accesses from site S_i to replica R_j of fragment F_j as $H_{cur}[S_i, R_j]$, while the old histogram is $H_{old}[S_i, R_j]$

D.5.2 Histogram Operations

This section presents algorithms for the different histogram operations.

Histogram Update

Every time a tuple in one of the local replicas is accessed, the corresponding histogram is updated. This is described in Algorithm 1. Although not included in the algorithms (to improve clarity), we normalize values before they are entered into the histogram. Assume a replica R_i of fragment F_i with $FVD(F_i) = F_i[min_i, max_i]$ and a tuple t_j with fragmentation attribute value v_j. We then record the value $v_j - min_i$. This means that histogram bucket numbers start at 0 regardless of the *FVD*.

Since this operation is performed very often, it is important that it is efficient. As described above, the value range of bucket number b_k is $[b_k \cdot W, (b_k + 1) \cdot W)$. We therefore need to determine b_k for a given fragmentation attribute value v_j and then increment its bucket value. The formula is $b_k = v_j / W$, which means that the computational cost is $O(1)$. Also, since histograms are kept in main memory, histogram updates do not incur any disk accesses.

If no bucket already exists for bucket number b_k, a new bucket must be constructed. This is the only time where the histogram gets more buckets, so after the update, the current number of buckets is checked against the upper bound MAX_B

Algorithm 1 Site S_i reads tuple t_j in replica R_j with fragmentation attribute value v_j. (Similar for writes.)

```
histogramUpdate(S_i, R_j, v_j):
  H_i ← H_cur[S_i, R_j]
  b_k ← v_j / W
  R_i[b_k] ← R_i[b_k] + 1
  if numberOfBuckets > MAX_B then
    increaseBucketWidth(R_j)
  end if
```

Algorithm 2 Increase bucket width W for histograms for replica R by factor Z_W.

```
increaseBucketWidth(R):
  for all S_i ∈ getActiveSites(R) do
    for all H_i ∈ H_cur[S_i, R] ∪ H_old[S_i, R] do
      H'_i ← ∅
      for all b_k ∈ H_i do
        b'_k = b_k / Z_W
        R'_i[b'_k] = R'_i[b'_k] + R_i[b_k]
        W'_i[b'_k] = W'_i[b'_k] + W_i[b_k]
      end for
      H_i ← H'_i
    end for
  end for
```

and bucket width is increased (and thus the number of buckets decreased) if we now have too many buckets.

Histogram Bucket Resizing

If at any time a tuple access occurs outside the range covered by the current buckets, a new bucket is made. If the upper bound of buckets, MAX_B, is reached, the bucket width W is increased and the histograms reorganized. We do this by multiplying W with a scaling factor Z_W. This factor is an integer such that the contents of new buckets are the sum of a number of old buckets. Increasing bucket width of course reduces the histogram accuracy, but it helps reduce both memory usage and processing overhead. Since we only store recent history, we may reach a point where the set of buckets in use becomes very small. If we can reduce bucket width to W/Z_W and still have fewer buckets than the upper bound, the histogram is reorganized by splitting each bucket into Z_W new buckets. This reorganization assumes uniform distribution of values inside each bucket, which is a common assumption [18]. Details are shown in Algorithm 2. Note that this is performed for both the current and old set of histograms in order to make them have the same bucket width, as this makes subsequent histogram accesses efficient. The function *getActiveSites*(R) returns the set of all sites that have accessed replica R.

Similarly, if we at any point use only a very low number of buckets, the bucket

widths can be decreased in order to make access statistics more accurate. This is described in Algorithm 3. Of special note is the expression $max(1, R_i[b_k]/Z_W)$. If a large bucket to be divided into smaller buckets contain only a few tuples, rounding can make $R_i[b_k]/Z_W = 0$, which would in effect remove the bucket (since only buckets containing tuples are stored). To prevent loss of information in this case, new buckets contain a minimum of 1 tuple.

Histogram Range Count

When retrieving access statistics from histograms, i.e., contents of buckets within a range, both current and old histograms are used. Since both histograms have the same bucket width and corresponding bucket numbers, retrieval is a straight summation of range counts from the two histograms and therefore very fast to perform. In order to count number of reads or writes from site S to replica R stored in buckets numbered $[b_{min}, b_{max}]$, the functions $histogramReadCount(S, R, b_{min}, b_{max})$ and *histogramWriteCount(S, R,* b_{min}*,* b_{max}*)* are used. In order to get the sum of range counts for writes from all sites, the function $histogramWriteCountAll(R, b_{min}, b_{max})$ is used.

Histogram Reorganization

As stated earlier, it is important that only the recent access history is used for replica evaluations in order to make it possible to adapt to changes in access patterns. This is achieved by having two sets of histograms, one current histogram H_{cur} that is maintained and one H_{old} which contains statistics from the previous period. Periodically the current H_{old} is replaced with the current contents of H_{cur}, and then H_{cur} is emptied and subsequently used for new statistics.

The only time buckets are removed from the histogram is during reorganization. It is therefore the only time that the number of buckets in the histogram can get so low that we can decrease the bucket width (thus creating more buckets) and still stay below the bucket number maximum MAX_B. This will be performed using $decreaseBucketWidth(R)$ described in Algorithm 3. The function performing the reorganization is in the following denoted $histogramReorganize(R)$.

D.5.3 Histogram Memory Requirements

It is important that the size of the histograms is small so that enough main memory is available for more efficient query processing and buffering. For every replica a site has, it must store two histograms for each active site accessing the fragment. Every bucket is stored as a $(b_k, R_i[b_k], W_i[b_k])$ triplet (note that sparse histograms are used, so that only buckets actually accessed are stored). Assuming b buckets and c active sites, the memory requirement for each replica is $2 \cdot c \cdot b \cdot sizeOf(bucket)$ or $O(b \cdot c)$. Since b has an upper bound MAX_B, memory consumption does not depend on fragment size or number of accesses, only on the number of active sites.

Algorithm 3 Decrease bucket width W for histograms for replica R by factor Z_W.

$decreaseBucketWidth(R)$:
  for all $S_i \in getActiveSites(R)$ **do**
    for all $H_i \in H_{cur}[S_i, R] \cup H_{old}[S_i, R]$ **do**
      $H'_i \leftarrow \emptyset$
      for all $b_k \in H_i$ **do**
        for $b'_k = 0$ to Z_W **do**
          $R'_i[b_k \cdot Z_W + b'_k] = max(1, R_i[b_k]/Z_W)$
          $W'_i[b_k \cdot Z_W + b'_k] = max(1, W_i[b_k]/Z_W)$
        end for
      end for
      $H_i \leftarrow H'_i$
    end for
  end for

D.6 Fragmentation and Replication

Our approach calls for three different algorithms. One for creating new replicas, one for deleting replicas and one for splitting and coalescing fragments. These will be described in the following sections.

These algorithms are designed to work together to dynamically manage fragmentation and replication of those fragments such that the overall communication costs are minimized. The communication cost consists of four parts: 1) remote writes, 2) remote reads, 3) updates of read replicas, and 4) migration of replicas (either in itself or as part of creation of a new replica).

Common for all three algorithms is that they seek to estimate the benefit from a given action based on available usage statistics. This is implemented using three cost functions, one for each algorithm. These functions are described in Section D.6.4.

D.6.1 Creating Replicas

This algorithm is run at regular intervals for each fragment of which a given site has a replica. The aim is to identify sites that, based on recent usage statistics, should be assigned a replica. If any such sites are found, replicas are sent to them, and the site holding the master replica is notified so that the new replicas can receive updates.

The algorithm for identifying and creating new replicas of replica R is shown in Algorithm 4. In the algorithm, a cost function (to be described in Section D.6.4) is applied for each remote site S_r that has read from to R. The result is a *utility value* that estimates the communication cost reduction achieved by creating a new replica at site S_r. All sites with positive utility value receive a replica. If no site has a positive utility, no change is made.

Note that, if desired, the number of replicas in the system can be constrained by having a limit on the number of replicas. This might be beneficial in the the context of massive read access to various sites.

Algorithm 4 Evaluate replica R for any possible new replicas. R is located on site S_l.

createReplica(R):

$b_{min} \leftarrow min(H_{cur}[S_l, R])$ {First bucket used}
$b_{max} \leftarrow max(H_{cur}[S_l, R])$ {Last bucket used}
$card_w \leftarrow histogramWriteCountAll(R, b_{min}, b_{max})$
for all $S_r \in getActiveSites(R)$ **do**
$card_r r \leftarrow histogramReadCount(S_r, R, b_{min}, b_{max})$
$utility \leftarrow w_{BE} \cdot card_r r - card_w - w_{FS} \cdot card(R)$
if $utility > 0$ **then**
$copyReplica(R, S_r)$ {Also notifies master replica}
end if
end for

Algorithm 5 Evaluate local replica R and decide if it should be deleted. R is located on site S_l.

deleteReplica(R):

$b_{min} \leftarrow min(H_{cur}[S_l, R])$ {First bucket used}
$b_{max} \leftarrow max(H_{cur}[S_l, R])$ {Last bucket used}
$card_w \leftarrow histogramWriteCountAll(R, b_{min}, b_{max})$
$card_l r \leftarrow histogramReadCount(S_l, R, b_{min}, b_{max})$
$utility \leftarrow w_{BE} \cdot card_w - card_l r$
if $utility > 0$ **then**
$deleteLocalReplica(R)$ {Also notifies master replica}
end if

D.6.2 Deleting Replicas

Since each fragment must have a master replica, only read replicas are considered for deletion. This algorithm evaluates all read replicas a given site has, in order to detect if the overall communication cost of the system would be lower if the replica were deleted. The details are shown in Algorithm 5. Again, a cost function is used to evaluate each read replica R. Any replica with a positive utility is deleted after the site with the master replica has been notified.

D.6.3 Splitting Fragments

The aim of the fragmentation algorithm is to identify parts of a table fragment that, based on recent history, should be extracted to form a new fragment and migrated to a different site in order to reduce communication costs (denoted *extract+migrate*). To avoid different fragmentation decisions made simultaneously at sites with replicas of the same fragment, this algorithm is only applied to master replicas.

More formally, assume a fragmentation $\mathcal{F}_{old}$ which includes a fragment F_i with $FVD(F_i) = F_i[min_i, max_i]$ having master replica R_i^m allocated to site S_i. Find a set of fragments $F_m, ..., F_n$ such that $\cup F_m, ..., F_n = F_i$ with $F_{new} \in F_m, ..., F_n$ and

Algorithm 6 Evaluate fragment F for any possible extract+migrates. R^m is the master replica of F and is currently located on site S_l

$refragment(F, R^m)$:
  $fragmentations \leftarrow \emptyset$
  for all $S_r \in getActiveSites(R^m)$ **do**
    for all $b_{min} \in H_{cur}[S_r, R^m], b_{max} \in H_{cur}[S_r, R^m]$ **do**
      $card_{rw} \leftarrow histogramWriteCount(S_r, R^m, b_{min}, b_{max})$
      $card_{lw} \leftarrow histogramWriteCount(S_l, R^m, b_{min}, b_{max})$
      $utility \leftarrow w_{BE} \cdot card_{rw} - card_{lw} - w_{FS} \cdot card(F)$
      if $utility > 0$ **and** $(max - min + 1) > fragmentMinSize$ **then**
        $fragmentations \leftarrow fragmentations \cup (S_r, min, max, utility)$
      end if
    end for
  end for
  $sort(fragmentations)$ {Sort on utility value}
  $removeIncompatible(fragmentations)$
  for all $(S_r, min, max, utility) \in fragmentations$ **do**
    $F_1, F_{new}, F_2 \leftarrow extractNewFragment(F, min, max)$
    $migrateFragment(F_{new}, S_r)$ {Migrates master replica}
    $updateReplicas()$
  end for
  $coalesceLocalFragments()$
  $histogramReorganize(R^m)$

master replica R^m_{new} allocated to site $S_k \neq S_i$ such that the communication cost $C_{total} = \sum C(A_i) + \sum C(RE_j)$ is lower than for $\mathcal{F}_{old}$.

The result of each execution can be either: 1) do nothing, i.e, the fragment is as it should be, 2) migrate the whole master replica, or 3) extract a new fragment F_{new} with $FVD(F_{new}) = F_{new}[min_{new}, max_{new}]$ and migrate its new master replica to site S_k. A decision to migrate the whole master replica can be seen as a special case of extract+migrate. In the discussion below, we therefore focus on how to find appropriate values for min_{new} and max_{new}. If a refragmentation decision is made, all sites with read replicas are notified so that they can perform the same refragmentation. This is necessary to enforce that all replicas of a given fragment are equal.

The algorithm for evaluating and refragmenting a given fragment F is presented in Algorithm 6. It evaluates all new possible fragments F_{new} and possible recipient sites S_r using a cost function. The result is a utility value that estimates the communication cost reduction from extracting F_{new} and migrating its master replica to S_r. Afterward, all *compatible* fragmentations with positive utility values are performed. Two fragmentations are compatible if their extracted fragments do not overlap. In case of two incompatible fragmentations, the fragmentation with the highest utility value is chosen. Note that no fragments with FVD less than *fragmentMinSize* will be extracted in order to prevent refragmentation from resulting in an excessive

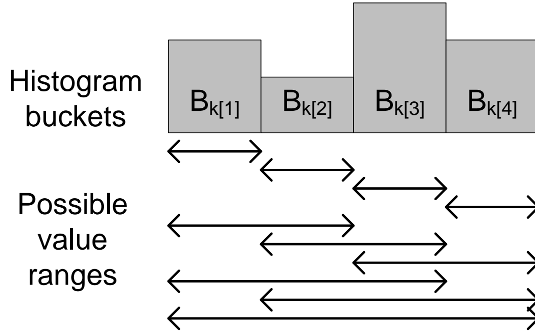


Figure D.3: Histogram with four buckets and corresponding value ranges.

number of fragments.

Given a fragment F_i with $FVD(F_i) = F_i[min_i, max_i]$, the size of the fragment value domain is then $width = max_i - min_i + 1$. Assume an extraction of a new fragment F_{new} such that $FVD(F_{new}) = F_{new}[min_{new}, max_{new}] \in FVD(F_i)$. If $FVD(F_{new})$ is assumed to be non-empty, i.e., $max_{new} > min_{new}$, then $width - 1$ possible values for min_{new} and max_{new} are possible. This means that $O(width^2)$ possible fragments F_{new} will have to be evaluated. This could easily lead to a prohibitively large number of F_{new} to consider, so some heuristic is required.

We reduce the number of possible fragments to consider based on the following observation: The basis for the evaluation algorithm is the access histograms described above. These histograms represent an approximation since details are limited to the histogram buckets. It is therefore only meaningful to consider $FVD(F_{new})$ with start/end-points at histogram bucket boundaries.

With b histogram buckets and $b \ll width$ as well as b having an upper bound, processing becomes feasible. The number of value ranges to consider is $b(b+1)/2 = O(b^2)$. An example of a histogram with four buckets and 10 possible $FVD(F_{new})$ is shown in Fig. D.3.

After the algorithm has completed, any adjacent fragments that now has master replicas on the same site are coalesced (denoted *coalesceLocalFragments*() in the algorithm). This helps keeping the number of fragments low. If two fragments are coalesced, the read replicas of those fragments must be updated as well. Some sites will likely have read replicas of only one of the fragments. These sites must either delete their replicas or get a replica of the fragment they are missing so coalescing can be performed on all replicas. Our heuristic is that we send the fragment which requires least communication cost to the sites missing that fragment. The remaining sites delete their local replicas.

Finally, old access statistics are removed from any remaining local master replicas using function *histogramReorganize*, as described in Section D.5.2.

D.6.4 Cost Functions

The core of the algorithms are the cost functions. The functions estimate the communication cost difference (or *utility*) between taking a given action (create, delete, split) and keeping the status quo. The basic assumption is that future accesses will resemble recent history as recorded in the access statistics histograms.

From Section D.3.3 the communication cost $C_{total} = \sum_i C(A_i) + \sum_j C(RE_j)$. Accesses can either be reads, writes or updates: $\sum_i C(A_i) = \sum_k C(AR_k) + \sum_l C(AW_l) + \sum_m C(AU_m)$. The recent history for fragment F consists of a series of accesses $SA = [A_1, ..., A_n]$. Each access A_i comes from a site S_o. The accesses from a given site S_o is $SA(S_o)$ where $SA(S_o) \subset SA$. Since we measure communication cost, local accesses have no cost, i.e., $\forall A_i, A_i \in SA(S_l) \Rightarrow C(A_i) = 0$.
The basic form of the cost functions is as follow:

$$utility = benefit - cost \tag{D.1}$$

Replica creation: The benefit of creating a new read replica on site S_r is that reads from site S_r will become local operations and thus have no network communication cost. The cost of creating a new replica is first that the new replica will have to be updated whenever the master replica is written to. The second part of the cost is the actual transfer of the replica to the new site. This gives the following utility function:

$$utilityCreate = card(SR(S_r)) - card(SU) - card(F) \tag{D.2}$$

where $card(SR(S_r))$ is the number of reads from remote site S_r, $card(SU)$ is the number of replica updates and $card(F)$ is the size of the fragment.

Replica deletion: When a read replica R at site S_l is deleted, the benefit is that replica updates will no longer have to be transmitted to S_l. The cost is that local reads from S_l to R will now become remote. Thus we get the following utility function:

$$utilityDelete = card(SU) - card(SR(S_l)) \tag{D.3}$$

Splitting fragments and migrating master replicas: As described earlier, the algorithm handles splitting by using the cost function on all possible value ranges for the fragment. Thus the aim of the cost function is limited to estimating when a master replica R should be migrated from S_l to a remote site S_r. The only way a migration of the master replica can affect the number of remote reads and updates in the system, is if S_r already has a read replica. However, since S_l does not know the usage statistics of any possible replica at S_r, we simplify the function by omitting this possibility. The benefit of a migration of the master replica to S_r is therefore that writes from S_r will become local operations. Similarly, the cost will be writes from S_l. In addition we must consider the cost of migrating in itself. Our utility function:

$$utilityMigrate = card(SW(S_r)) - card(SW(S_l)) - card(F) \tag{D.4}$$

Cost function weights: While these equations are expressions of possible communication cost savings from different actions, they cannot be used quite as they

are in an actual implementation. There are a couple of issues. First, SW, SR and SU by design include only the recent history and cardinality values are therefore dependent on how much history we include. On the other hand, $card(F)$ is simply the current number of tuples in the fragment and thus independent on history size. We therefore scale $card(F)$ by a *cost function weight* w_{FS}. This weight will have to be experimentally determined and optimal value will depend on how much the usage history includes.

The second problem is stability. If we allow, e.g., migration when the number of remote accesses is just a few more than the number of local accesses, we could get an unstable situation where a fragment is migrated back and forth between sites. This is something we want to prevent as migrations cause delays in table accesses and indices may have to be recreated every time. To alleviate this problem, we scale the *benefit* part of the cost functions by $w_{BE} \in [0..1]$. For migrations, $w_{BE} = 0.5$ means that there will have to be 50 % more remote accesses than local accesses for migration to be considered, i.e., for the utility to be positive (disregarding fragment size).

By including w_{FS} and w_{BE} we get the following cost functions:

$$utilityCreate = w_{BE} \cdot card(SR(S_r)) - card(SU) - w_{FS} \cdot card(F) \tag{D.5}$$

$$utilityDelete = w_{BE} \cdot card(SU) - card(SR(S_l)) \tag{D.6}$$

$$\begin{aligned} utilityMigrate = w_{BE} \cdot card(SW(S_r)) - card(SW(S_l)) \\ - w_{FS} \cdot card(F) \end{aligned} \tag{D.7}$$

Different values for the two cost function weights are evaluated experimentally in the Evaluation section below.

D.7 Evaluation

In this section we present an evaluation of our approach. We aim to investigate different dynamic workloads and the communication cost savings our algorithms can achieve. Ideally, we would have liked to do a comparative evaluation with related work. However, to the best of our knowledge, no previous work exists that do continuous dynamic refragmentation and replication based on reads and writes in a distributed setting. Instead, we compare our results with a no-fragmentation and an optimal fragmentation method (where applicable).

The evaluation has three parts. First we examine the results from running a simulator on four workloads involving just two sites. These workloads have been designed to highlight different aspects, such as fragmentation, replication and changing access patterns. We have kept them as simple as possible to make it easier to analyze the results qualitatively. For the second part of the evaluation, we do simulations using two highly dynamic workloads involving more sites, providing a more realistic setting. The third part consists of experiments on an implementation in a distributed database system.

D.7.1 Experimental Setup

For the evaluation, we implemented a simulator which allows us to generate distributed workloads, i.e., simulate several sites all performing tuple reads and writes with separate access patterns. In all presented simulation results, the fragmentation and replication decision algorithms were run every 30 seconds. All simulations have been run 100 times, and we present the average values. For each simulated site, the following parameters can be adjusted:

- Fragmentation attribute value interval: minimum and maximum values for the accesses from the site.
- Access distribution: either uniform or hot spot (10 % of the values get 90 % of the accesses).
- Average rate of tuple accesses in number of accesses per minute. We use a Poisson distribution to generate accesses according to the frequency rate.
- Access type: reads, writes or a combination of both.

Values for these parameters need not be constant, but can change at any point for any site in the workload. In our simulations, we have used maximum histogram size of $MAX_B = 100$ buckets, and each table has one fragment with no read replicas when a simulation starts. We also tested with 1000 buckets, but this provided negligible benefits for our workloads.

Unlike most of the relevant previous work, our method tightly integrates fragmentation allocation and replication. Therefore, it does not make much sense comparing against techniques that only perform one of the tasks. Instead, we use the following two fragmentations methods to act as baselines for comparison. The first is a baseline where *the table is not fragmented or replicated at all.* The table consists of a single fragment with its master replica permanently allocated to the site with the largest total number of accesses. This is what would happen in a database system that does not use fragmentation or replication (e.g., to simplify implementation and configuration), at least given that workloads were completely predictable. Since there is no replication, there are no communication costs from migrations either.

The second allocation method we compare against, is *optimal fragmentation.* Here we assume full knowledge about future accesses. Each table is (at runtime) fragmented and the fragments are migrated and/or replicated to the sites which would minimize remote accesses.

It should be noted that both these fragment allocation alternatives assume advance knowledge about the fragmentation attribute value interval, distribution, frequency and type of accesses, none of which are required for our dynamic approach.

D.7.2 Workloads Involving Two Sites

In this section, we present results from four workloads, each with two sites (S_1, S_2). These two sites accessed 25000-50000 tuples each. Early testing showed that 25000

Table D.4: Two-site workloads.

Workload no.	Access	Distribution	Rate	Purpose
1	Write	S_1:Uniform, S_2:Hot spot	Low	Detect hot spots
2, first half	Write	S_1:Hot spot, S_2:Uniform	Low	Detect distribution change
2, second half	Write	S_1:Uniform, S_2:Hot spot	Low	
3	S_1:Read, S_2:Write	Uniform	S_1:High, S_2:Low	Make read replica
4, first half	S_1:Read, S_2:Write	Uniform	S_1:High, S_2:Low	Change replica pattern
4, second half	S_1:Write, S_2:Read	Uniform	S_1:Low, S_2:High	

tuples was more than enough to reach a stable situation. Only two sites were used for these workloads in order to make it easier to analyze the results. Each workload was therefore designed with a specific purpose in mind.

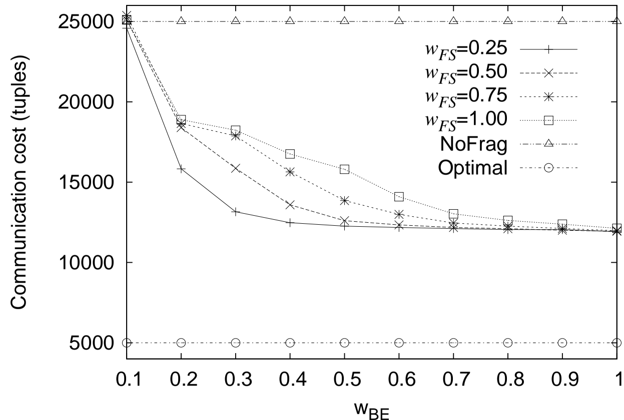
The fragmentation attribute value intervals for the two sites were designed so that they overlapped completely. Two rates were used, a high rate of 6000 accesses per minute and a low rate of 3000 accesses per minute. For workload 1 and 3, the workload was constant for both sites, while 2 and 4 switched workload parameters halfway through. Workloads 2 and 4 serve as examples of dynamic workloads where access patterns are not constant and predictable. The results from these workloads should illustrate if our approach's ability to adjust fragmentation and replication at runtime result in communication cost savings. The four workloads are detailed in Table D.4.

Workload 1: In this workload, one of the sites has 10 hot spots while the other has uniform access distribution. Ideally, these 10 hot spots should be detected and migrated while the remainder should be left on the uniform access site. This case is similar to the one presented in Fig. D.1. Fig. D.4(a) shows results for workload 1 with different values for w_{BE} and w_{FS}. Communication costs for *no-fragmentation* and *optimal fragmentation* are also shown.

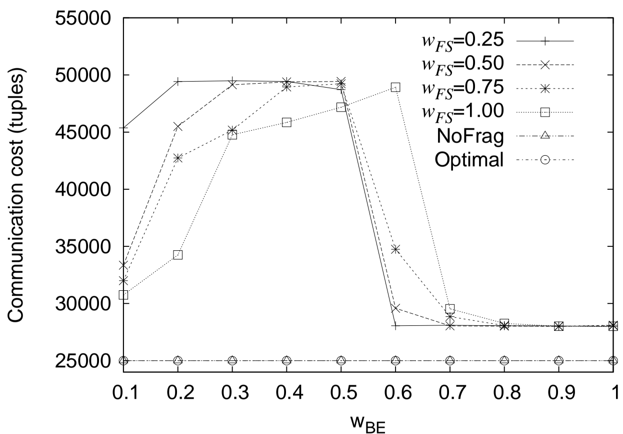
For this workload, the majority of the communication cost comes from remote writes, i.e. when the extract+migrate algorithm is very conservative on migrating the hotspots from S_1 to S_2. High values of w_{FS} cause the algorithm to overestimate the cost of migration while low values of w_{BE} cause the benefit to be undervalued. This combination thus almost reduces to the no-fragmentation case. For lower values of w_{FS} and higher values of w_{BE}, refragmentation decisions are made earlier and the result is comparable to optimal fragmentation.

Workload 2: This is a dynamic version of workload 1, with the two sites switching

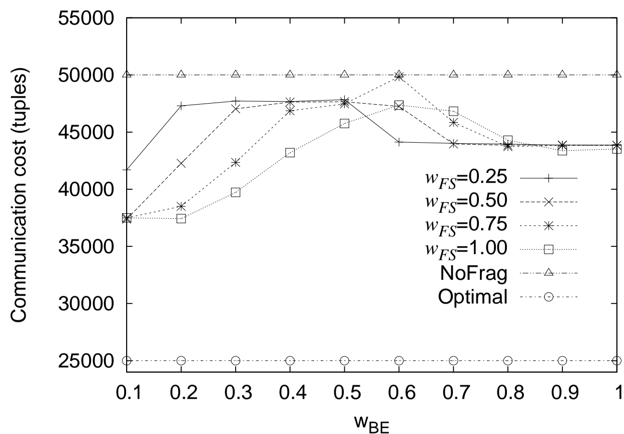
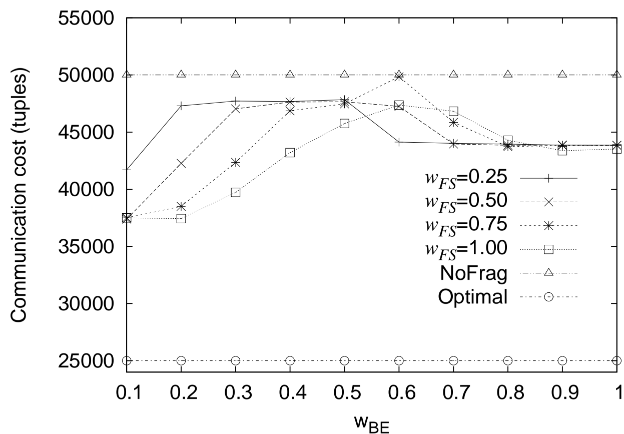
(a) Two-site workload 1.



(b) Two-site workload 2.



(c) Two-site workload 3.

(d) Two-site workload 4.

Figure D.4: (a) Results from two-site workload 1. (b) Results from two-site workload 2. (c) Results from two-site workload 3. Note that NoFrag and Optimal are equal for this workload, at 25000 tuples. (d) Results from two-site workload 4.

access patterns halfway through. The simulation results for this workload are shown in Fig. D.4(b).

Results here are similar to workload 1, but with an extra overhead from detecting the access pattern change. This overhead is larger than for workload 1 simply because at the time the workload changes, the recent history is filled with the old workload and it takes a while for the new workload to dominate. The worst result is again similar to no-fragmentation.

Workload 3: This workload has one site writing while the other site reads at twice the rate. Ideally the site that writes should get the master replica, while the other site gets a read replica. Results from workload 3 are shown in Fig. D.4(c).

The most important factor for the communication cost of this workload is whether a read replica is created on S_2. For low values of w_{BE}, the benefit of such a replica is undervalued and it is never created leading to poor results. Changes in w_{FS} only delay replica creation slightly and therefore has comparatively little influence. The exception is where high w_{FS} and low w_{BE} together prevent any migrations from happening, giving similar results to no-fragmentation. No-fragmentation does quite well here as it allocates the fragment to the site with the highest number of accesses which is also the optimal solution.

Workload 4: Similar to workload 3, except the two sites change behavior halfway through the workload. What we would like to see is a deletion of the read replica, migration of the master replica and a subsequent creation of a new read replica. Results from workload 4 are shown in Fig. D.4(d).

The results are somewhat similar to workload 3. The largest difference is the overhead from detecting the workload change (similar to that of workload 2). For low values of w_{BE}, remote reads are the dominant cost since no replica is created. For higher values, a replica is created and remote updates dominates. No-fragmentation is now much worse since it does not adjust to the change.

Detailed results for all four workloads with $w_{BE} = 0.9, w_{FS} = 0.50$ are shown in Table D.5. This table lists the number of remote accesses, migrations, fragments at the end of the run and the number of tuples transferred during migrations. The communication cost is the sum of remote accesses and tuples transferred. The final two columns shows the communication cost from the no-fragmentation and optimal allocation methods. Average results for the four workloads using the same cost function weight values are shown in Fig. D.5.

D.7.3 Workloads Involving Several Sites

This section presents the results from two workloads involving 20 *active* sites each (i.e., the actual system can consist of a much larger number of sites, however only 20 sites simultaneously access the actual table during the simulation). The first of these workloads is intended to resemble a distributed application which have separate read and write phases, e.g., a grid application.

We have modeled the read phase as follows: A site uniformly accesses an random interval that constitutes 10% of the table. Between 30.000 and 60.000 reads are performed at an access rate of 2000 to 4000 reads a minute. Values for the interval,

Table D.5: Detailed results, $w_{BE} = 0.9, w_{FS} = 0.50$.

Workl. no.	Re. writes	Re. reads	Re. updates	Migr.	Frag.	Tuples	Comm. cost	No frag.	Optimal
1	6229	0	0	10	20	46	6275	25000	5000
2	11145	0	0	53	47	860	12005	25000	5000
3	984	3154	22476	2	1	1385	27999	25000	25000
4	4009	6374	30310	44	21	3173	43866	50000	25000

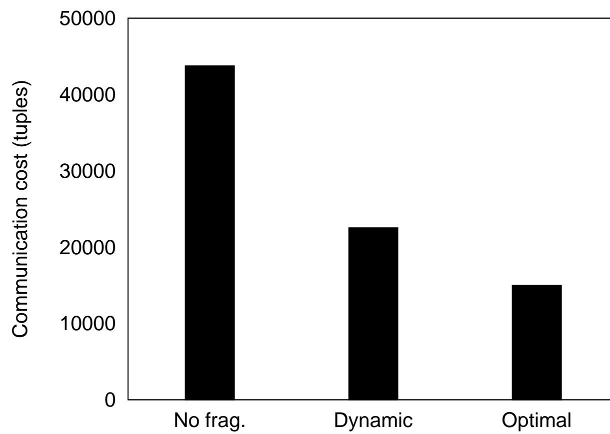


Figure D.5: Comparative results from two-site workloads.

number of reads and rate are drawn randomly at the start of each phase.

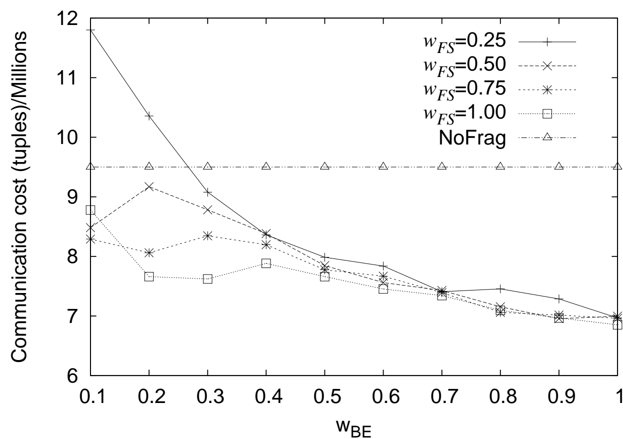
After the read phase has completed, a write phase follows. Here the site uniformly accesses a random interval 1% of the size of the table. Anywhere from 20.000 to 40.000 tuples are written at a rate of 2000 writes a minute. After the write phase has completed, a new read phase is initiated (and so on) until the site has accessed 500.000 tuples. With 20 sites, this gives a complete workload consisting of 10 million accesses. Also note that due to the random parameters, two different sites will generally not be in the same phase.

Comparative evaluation is more difficult for this workload than for those previously presented. The no-fragmentation method is still usable, but less realistic as the fixed non-fragmented master replica easily can become a bottleneck for remote writes and updates. The optimal fragmentation method is more problematic. With 10 million accesses each run and no clear access pattern, a very large number of fragmentations, migrations and replica allocations would have to be evaluated to find the optimal dynamic solution. Further, the highly random nature of this workload means that a fragmentation and replica allocation that are optimal for one run, will not be optimal for another. The optimal fragmentation method would therefore have to be recomputed for each run. For these reasons we found optimal fragmentation infeasible and omitted it from this part of the evaluation.

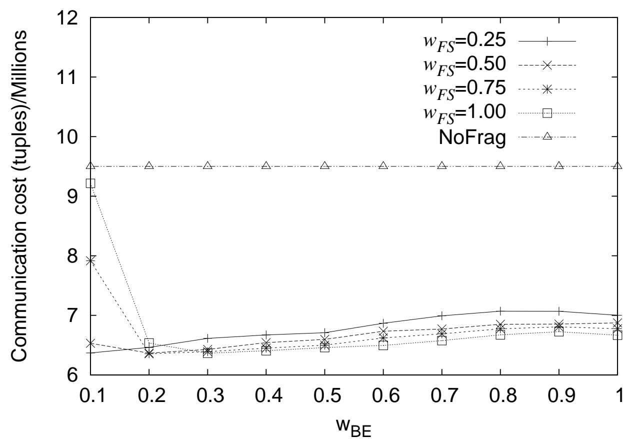
The results are shown in Fig. D.6(a). With ten times as many sites as for earlier workloads, having too many replicas becomes a much more important issue due to the number of update messages needed to keep all the replicas consistent. This is what causes very poor results with a combination of low w_{BE} and low w_{FS}. The low w_{FS} underestimates the cost of creating a read replica while low w_{BE} makes it hard to delete it later. This leads to an excessive number of replicas and poor performance from the high number of updates needed. Due to the highly dynamic nature of this workload, high values of w_{BE} work well as they make the algorithms take action earlier. Since the number of writes is low and confined to narrow intervals of the table, fragment sizes stay small and thus the w_{FS} value is of little importance.

The second multi-site workload is intended to resemble a more general usage pattern where each site does not have distinct read and write phases, but rather a single phase that includes both. We have modeled it as follows: A site uniformly accesses a random interval that constitutes 10% of the table. Each of these accesses can be either a read (80%) or a write (20%). The access rate is from 2000 to 4000 accesses a minute, and the phase lasts between 30.000 and 60.000 accesses. After the phase has completed, it restarts with new sets of parameters randomly drawn. As for the last workload, this continues until 500.000 accesses have been made from each site. The simulation results are shown in Fig. D.6(b).

Similar to the grid application workload, the creation and deletion of read replicas are the most important factors influencing the results. Low values of w_{BE} make the algorithms act conservatively, both when creating and deleting replicas. This leads to remote reads dominating the communication cost. For higher values of w_{BE}, more replicas are created giving fewer remote reads but more updates. For this workload, these two factors tended to balance each other out, giving similar communication costs for a wide selection of cost function weight values. While there are separate

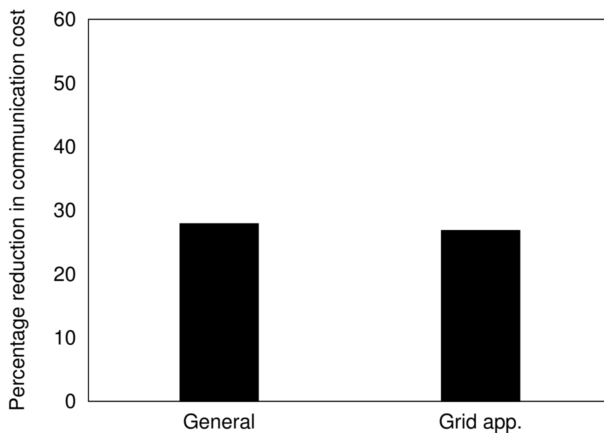


(a) Grid application workload.

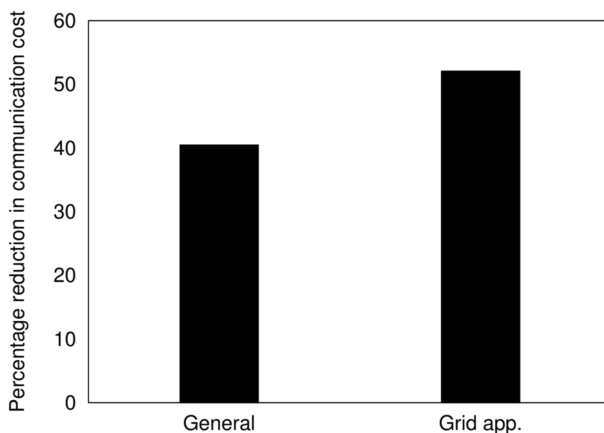


(b) General workload

Figure D.6: (a) Results from grid application workload. (b) Results from general workload.



(a) Multi-site workloads in simulations.



(b) Multi-site workloads in DASCOSA-DB.

Figure D.7: (a) Comparative results from simulations with multi-site workloads, showing reduction in communication cost relative to the no-fragmentation method. (b) Comparative results from multi-site workloads using DYFRAM implemented in DASCOSA-DB, showing reduction in communication cost relative to the no-fragmentation method.

Table D.6: Tuples transferred during multi-site workloads in simulations and implementation in DASCOSA-DB.

	Simulation			Implementation		
Workload	No frag.	DYFRAM	Reduction	No frag.	DYFRAM	Reduction
General	9.5 mill.	6.85 mill.	27.9%	100.000	59519	40.5%
Grid app.	9.5 mill.	6.95 mill.	26.8%	100.000	47921	52.1%

write phases in the grid application workload that each only accessed 1% of the table, writes in this workload were interleaved with reads and accessed a much larger part of the table (for a given phase). This workload also had a smaller fraction of the accesses as writes. These three factors caused the splitting algorithm to create smaller fragments which meant that w_{FS} had little impact on the results.

Comparative results for the two multi-site workloads using $w_{BE} = 0.9$ and $w_{FS} = 0.50$, are shown in Fig. D.7(a) and Table D.6.

D.7.4 Implementation of DYFRAM in DASCOSA-DB

In this experiment, DYFRAM was implemented in the DASCOSA-DB distributed database system [16] in order to verify simulation results. The workloads tested are similar to the grid and general workloads presented in Section D.7.3, but have been scaled down a bit for practical reasons.

The grid workload has read phases of 6.000–12.000 accesses. Each phase uniformly accesses a random 5% interval of the table. Write phases do 4.000–8.000 writes to a random 0.5% interval of the table. There is no delay between accesses. As soon as a site finishes one phase, it starts on the next, alternating between read and write phases. The experiments are done with 6 sites, each issuing 20.000 accesses, i.e., a total of 120.000 accesses. Half of the sites start in a read phase, while the other half starts in a write phase. Due to the different phase lengths, this pattern will change several times during the experiment.

The general workload is scaled with the same factors, giving phases of 6.000–12.000 accesses to 5% of the table. 80% of these are read accesses and 20% are write accesses. Each of the 6 sites issues 20.000 accesses, resulting in a total of 120.000 accesses.

The refragmentation algorithm is run every 30 seconds with $w_{FS} = 0.2$ and $w_{BE} = 0.95$, which should give a quite aggressive use of refragmentation and replication. As explained in Section D.6.4, the weights were found experimentally by testing on a shorter workload, consisting only of a few thousand accesses. The results are compared against the no-fragmentation method. Each experiment is repeated a number of times with different random seeds.

The results from both workloads and the no-fragmentation method are shown in Fig. D.7(b) and Table D.6. We see that the results are similar to those from the simulations. For the general workload, communication costs are reduced by more than 40% compared to the no-fragmentation method. The costs of the grid

workload is reduced by more than 50%. Clearly, the cost of replication is made up for by converting remote accesses to local accesses. Around 20% of the tuples transferred are caused by fragments moving around. The ratio of read vs. write accesses varies more, with the grid workload generally having higher write costs and the general workload having higher read costs.

The results do not vary much between each run, and small changes in w_{FS} and w_{BE} do not change the results much. The length of each phase will affect the cost savings, but even if phases are only half as long, communication costs are 25% below the no-fragmentation method.

D.8 Conclusions and Further Work

In distributed database systems, tables are frequently fragmented and replicated over a number of sites in order to reduce network communication costs. How to fragment, when to replicate and how to allocate the fragments to the sites are challenging problems that has previously been solved either by static fragmentation and allocation, or based on the analysis of a priori known queries. In this paper we have presented *DYFRAM*, a decentralized approach for dynamic table fragmentation and allocation in distributed database systems, based on observation of the access patterns of sites to tables. To the best of our knowledge, no previous work exists that perform the combination of continuous refragmentation, reallocation, and replication in a distributed setting.

Results from simulations show that for typical workloads, our dynamic fragmentation approach significantly reduces communication costs. The approach also demonstrates well its ability to adapt to workload changes. In addition to simulations, we have also implemented DYFRAM in the DASCOSA-DB distributed database system, and demonstrated its applicability in real applications.

Future work include exploring adaptive adjustment of the cost function weights as well as better workload prediction based on control theoretical techniques. We also intend to develop a variant of our approach that can be used in combination with static query analysis in order to detect periodically recurring access patterns.

Bibliography

[1] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of SIGMOD 2006*, 2006.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, 2004.

[3] I. Ahmad et al. Evolutionary algorithms for allocating data in distributed database systems. *Distributed and Parallel Databases*, 11(1):5–32, 2002.

[4] P. M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 13(3):263–304, 1988.

[5] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of SoCC '10*, 2010.

[6] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proceedings of ICDE*, 2007.

[7] A. Brunstrom, S. T. Leutenegger, and R. Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Proceedings of CIKM '95*, 1995.

[8] B. Ciciani, D. Dias, and P. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):247–261, Jun 1990.

[9] G. Copeland et al. Data placement in Bubba. In *Proceedings of SIGMOD 1988*, 1988.

[10] A. L. Corcoran and J. Hale. A genetic algorithm for fragment allocation in a distributed database system. In *Proceedings of SAC'94*, 1994.

[11] T. Didriksen, C. A. Galindo-Legaria, and E. Dahle. Database de-centralization - a practical approach. In *Proceedings of VLDB 1995*, 1995.

[12] D. Donjerkovic, Y. E. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proceedings of ICDE*, 2000.

[13] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of DOLAP 2004*, 2004.

[14] B. Gavish and O. R. L. Sheng. Dynamic file migration in distributed computer systems. *Commun. ACM*, 33(2):177–189, 1990.

[15] T. Hara and S. K. Madria. Data replication for improving data accessibility in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(11):1515–1532, 2006.

[16] J. O. Hauglid, K. Nørvåg, and N. H. Ryeng. Efficient and robust database support for data-intensive applications in dynamic environments. In *Proceedings of ICDE*, 2009.

[17] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of VLDB 1990*, 1990.

[18] Y. Ioannidis. The history of histograms (abridged). In *Proceedings of VLDB 2003*, 2003.

[19] M. Ivanova, M. L. Kersten, and N. Nes. Adaptive segmentation for scientific databases. In *Proceedings of ICDE 2008*, 2008.

[20] S. Menon. Allocating fragments in distributed databases. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):577–585, 2005.

[21] A. Mondal, S. K. Madria, and M. Kitsuregawa. CADRE: A collaborative replica allocation and deallocation approach for mobile-p2p networks. In *Proceedings of IDEAS 2006*, 2006.

[22] A. Mondal, K. Yadav, and S. K. Madria. EcoBroker: An economic incentive-based brokerage model for efficiently handling multiple-item queries to improve data availability via replication in mobile-p2p networks. In *Proceedings of DNIS 2010*, 2010.

[23] P. Padmanabhan, L. Gruenwald, A. Vallur, and M. Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *VLDB J.*, 17(5):1143–1164, 2008.

[24] J. Rao et al. Automating physical database design in a parallel database. In *Proceedings of SIGMOD 2002*, 2002.

[25] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1):29–56, 1985.

[26] D.-G. Shin and K. B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Trans. Software Eng.*, 17(9):872–883, 1991.

[27] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in mariposa. In *Proceedings of ICDE 1996*, 1996.

[28] M. Stonebraker et al. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[29] A. Tamhankar and S. Ram. Database fragmentation and allocation: an integrated methodology and case study. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 28(3):288–305, May 1998.

[30] T. Ulus and M. Uysal. Heuristic approach to dynamic data allocation in distributed database systems. *Pakistan Journal of Information and Technology*, 2(3):231–239, 2003.

[31] G. Weikum et al. The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5):381–432, 1994.

[32] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proceedings of PODS'92*, New York, NY, USA, 1992. ACM.

[33] E. Wong and R. H. Katz. Distributing a database for parallelism. *SIGMOD Rec.*, 13(4):23–29, 1983.

[34] D. C. Zilio et al. DB2 design advisor: integrated automatic physical database design. In *Proceedings of VLDB 2004*, 2004.

Paper E

Site-Autonomous Distributed Semantic Caching

Norvald H. Ryeng, Jon Olav Hauglid and Kjetil Nørvåg.
In *Proceedings of SAC*, 2011.

Abstract

Semantic caching augments cached data with a semantic description of the data. These semantic descriptions can be used to improve execution time for similar queries by retrieving some data from cache and issuing a remainder query for the rest. This is an improvement over traditional page caching, since caches are no longer limited to only base tables but are extended to contain intermediate results. In large-scale distributed database systems, using a central server with complete knowledge of the system will be a serious bottleneck and single point of failure. In this paper, we propose a distributed semantic caching method where sites make autonomous caching decisions based on locally available information, thereby reducing the need for centralized control. We implement the method in the DASCOSA-DB distributed database system prototype and use this implementation to do experiments that show the applicability and efficiency of our approach. Our evaluation shows that execution times for queries with similar subqueries are significantly reduced and that overhead caused by cache management is marginal.

E.1 Introduction

Large, distributed systems often use site autonomy as a way to reduce communication costs, allowing sites to make their own decisions and rely more on locally available information and less on information that must be fetched from their neighbors. In addition, if we can allow some of the housekeeping information to be slightly outdated without affecting the query results, further decoupling of sites is possible.

Caching is one aspect of a query processing system that lends itself to autonomous decisions. Each site can cache the data it needs to speed up its own processing, without coordinating with other sites first. In distributed database systems, there is a choice of either shipping data to the sites where queries are processed or shipping queries to the sites where data are stored. Caching is possible and useful in both types of systems, but the nature of these systems provide different caching opportunities and call for different caching solutions. In this paper, we adapt the idea of semantic caching, taken from data shipping systems, and present a new method for semantic caching for a large query shipping system.

The main idea of semantic caching is to tag the cached data items with semantic information, typically predicates used in select queries. By looking at the tags, subsequent queries can identify cached items that can replace parts of the query. The data that is not in cache is fetched by a remainder query, and together the remainder query and the cached query provides the answer to the original query.

One of the challenges that are introduced when semantic caching is moved into a system of autonomous sites is that no single site has full knowledge of the query workload. When queries can enter the system from any site, and each site processes only a small part of each query before the result is shipped off to the next site, no single site has the complete picture of the query workload. This limits the metrics available for caching algorithms, but as we demonstrate, it is still possible to make globally sound caching decisions.

Table E.1: Symbols.

Symbol	Description
S	Site
T	Table T
T_i	Fragment i of table T
n	Algebra node
N	Algebra tree
N_n	Subtree rooted at n
c	Cache entry
C	List of cache entries
$\mathcal{C}$	One site's cache
ts	Timestamp
$query(c)$	Query representation of cache entry

By building semantic caches of intermediate results on the sites where these results are produced, subsequent similar queries can benefit from retrieving some of their data from cache and issuing remainder queries to perform the rest of the operations. Our method builds a globally accessible, distributed cache based on autonomous sites. Whereas in traditional semantic caching each site has its own local cache that is not shared with other sites, our method gives sites access also to the cache entries elsewhere in the network.

With semantic caching comes the possibility of making caching decisions based on more than access statistics. Richer caching algorithms can be defined that inspect the semantics and decide to cache data that is not the most frequently used, but that will give a higher performance gain when used. Even if the join of two tables is a less frequent subquery than the tables themselves, more time may be saved if the result of the join is cached than if the tables are cached. By making sites autonomous, we also open up for the possibility of using different caching algorithms on different sites.

The contributions of this paper is as follows: We present a new method for semantic caching of intermediate results in a distributed database system, using autonomy to increase scalability. We demonstrate how this caching method reduces query execution time with almost no overhead. We also demonstrate that the more advanced cache replacement policies that are possible with a semantic cache give considerable improvements over traditional LRU. Experimental evaluation of the costs and benefits of our semantic caching method is done by implementing it in the DASCOSA-DB [9] distributed database prototype.

The rest of this paper is organized as follows. We start with a review of related work in Section E.2. Section E.3 describes the system setting. Our caching method is described in detail in Section E.4. Section E.5 describes our experiments and results, and we conclude the paper and outline future work in Section E.6.

E.2 Related Work

Semantic caching [7] and predicate-based caching [13] augment cached data with a semantic description of the data. The benefits of semantic caching include low overhead and reduced network traffic [11, 18]. Cache tables [1, 3, 16] are somewhat similar to semantic caching, but only caches tables, not intermediate or final results of queries. Semantic caching has also been applied to deductive databases [4] and web querying systems [6, 15]. Common to all these systems are that they are built for a single query entry point to the system.

There are several approaches to filling caches. Cache investment [14] aims to optimize for the future by deliberately executing suboptimal queries to generate cache entries that have a higher hit rate. Caching of time-consuming operations has also been studied for single values that are duplicated in a result [10]. Identification of candidates for caching and insertion as cache nodes at different levels in the algebra tree [8] is often done during query planning.

Cache entries may exist with similar, but not exactly the same, data to what has been requested. Such cache entries may be transformed to match the request [2]. This increases the cache hit rate for workloads with many similar queries. View materialization [5, 17] is a kind of explicit caching requiring manual intervention.

Our approach is similar to that of PeerOLAP [12] in that sites operate under a large degree of autonomy and make local caching decisions. However, while PeerOLAP uses broadcast messages to locate caches, our approach retrieves information on existing cache entries from a distributed catalog service. This way we avoid flooding the network on each query.

E.3 Preliminaries

In this section we describe the system and query model used in the rest of this paper. The symbols used in the paper are found in Table E.1. We have implemented our caching method in the DASCOSA-DB distributed database system prototype. Our caching method is general and does not put many limitations on the underlying system, but some details of the implementation are dependent on details of the underlying system.

E.3.1 System Model

The system consists of a number of sites, each site S_i being a single computer node or a parallel system acting as a single entity seen from other sites. All sites can store relational tables, and these may be horizontally fragmented. Fragment i of table T is denoted T_i.

The sites in the system are autonomous, and the only requirement is that they have a common protocol for execution of queries and metadata management. This means that some sites may choose not to cache, and those that do cache may choose different cache replacement policies. This makes it possible to include sites under

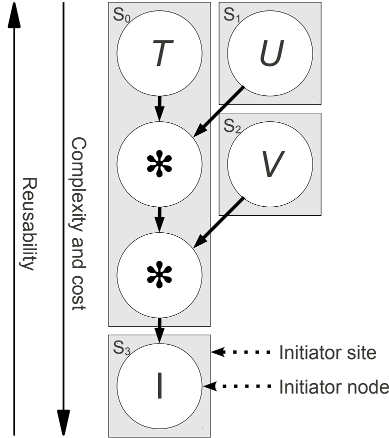


Figure E.1: Example query from unmodified DASCOSA-DB.

different administrative domains, allowing interoperability while at the same time allowing each administrative domain to remain autonomous.

Metadata management, including information on where data is stored, is performed through a common catalog service. This catalog service is itself assumed to be fault tolerant. It can be realized in a number of ways. For example, in DASCOSA-DB, the catalog service is realized by a distributed hash table where all sites participate. The organization of the catalog is not important to our caching method, but our implementation relies on some implementation details of the catalog service.

E.3.2 Query Model

We assume queries are written in some language that can be transformed into relational algebra operators, for example SQL. These algebra operators constitute an algebra tree for the query, and each subtree of the query tree is a subquery. Throughout this paper we will use the terms subtree and subquery interchangeably.

Queries may arrive from any site of the system. The site that introduces a query to the system, called the initiator site for that query, becomes the coordinator for that query. When a query is entered at one site, this site becomes the initiator site for that query. The initiator site decomposes the query into an algebra tree, e.g., as the one shown in Figure E.1. The example query accesses the three tables T, U and V located at sites, S_0, S_1 and S_2, respectively. Query processing is distributed between these three sites and the initiator site for the query, S_3. When the query planner has assigned each algebra node N_i to a site, S_i, the query is shipped to these sites using Algorithm 7.

When query processing starts, each node of the algebra tree produces an intermediate result that is shipped to the parent (or *downstream*) node. Our distributed

Algorithm 7 Stepwise transmission of algebra tree.

At site S_i, after receiving N_i:

$n_i \leftarrow root(N_i)$
for all $n_c \in children(n_i)$ **do**
  $N_c \leftarrow subtree(n_c)$
  $S_c \leftarrow getAssignedSite(n_c)$
  $Send(N_c, S_c)$
end for

semantic caching method caches these intermediate results, such as the result of $T * U$, and reuses them in subsequent queries. This can save significant amounts of processing. The cumulative cost and complexity of the intermediate results increases towards the root, but reusability decreases. Caching the result of nodes close to the root of the tree means we save more work when we get a cache hit, but cache hits are more frequent for intermediate results closer to the leaves.

E.4 Distributed Semantic Caching

In order to implement semantic caching, we modify the localization, dissemination and processing steps of DASCOSA-DB, and add a fourth: cache registration. These modifications and extensions are described in the following sections.

E.4.1 Query Localization

After query decomposition, the query is represented as a tree N of algebra operator nodes. This is given as input to the query localization step.

The initiator site has to do catalog lookups for all tables referenced by N. This is done by requesting from the catalog service a list of table fragments T_i and the sites S_{T_i} on which they are located. In DASCOSA-DB, the request for information about one table is handled by one site of the distributed catalog service. That site knows of all fragments of that table. We extend the information stored at the catalog site to also include an index of some, but not necessarily all (see Section E.4.4), cache entries of intermediate results involving that table.

We also extend the catalog lookup request by piggybacking a representation of N onto the request messages, as shown in Figure E.2. The catalog service site responds with its normal result of table fragments and their locations and adds an additional list $C = \{c_1, c_2, c_3, \ldots\}$ of cache entries it knows of. The extended reply message is shown in Figure E.3. Each entry $c = \langle N_c, S_c, ts_c \rangle$ describes the cached query N_c, the site S_c that stores the cache entry and the timestamp ts_c of the entry. All entries in C are relevant cache entries, by which we mean entries that can be used to answer the query.

After receiving all lookup replies, we have accumulated information on all relevant cache entries for N. The optimizer may decide to rewrite the query to use some cache entries that do not exactly match a subtree of the current plan. Due to space

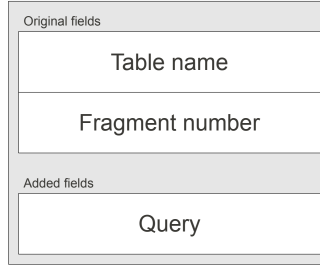


Figure E.2: Extended lookup message.

constraints, we refer to [7] and [11] for more details on how queries are transformed to take advantage of semantic caches.

After caches have been found and the query plan has been adapted, the next step is to assign each node n of N to a site. Leaf nodes are table accesses and are assigned to the sites that store the corresponding table fragments. Normally, DASCOSA-DB assigns an operator node to the same site as one of its operands. We extend this algorithm to exploit cached data. By looking at the subquery N_n rooted at n and the list of cache entries, if $\exists\, c : N_c = N_n$, n is assigned to S_c. If there are more than one cache entry for N_n, the localizer chooses one. Nodes that are not found in cache are assigned to sites using the normal query localization algorithm.

The initiator site does not actually decide whether to use a cache entry. It only assigns algebra nodes to sites where the catalog service says there are matching cache entries. Sites are autonomous in caching decisions, so a site that was intended by the initiator to deliver data from cache may have replaced the cache entry when the query arrives, making it necessary to process the query in full.

When all nodes of the query have been assigned to a site, the initiator site starts shipping out the query to the rest of the sites participating in resolving it.

E.4.2 Query Dissemination

The query N is shipped stepwise to the participating sites, using a modified version of Algorithm 7. The initiator site assigns $root(N)$ to itself, and then for all $n \in children(root(N))$ sends out N_n to S_n, which again send out the subtrees of the nodes they receive, etc. This continues until the leaf nodes, i.e., table access nodes, are received by the sites that store the corresponding tables. The timestamps ts_{T_i} of all table fragments referenced by nodes in a subtree are piggybacked onto that subtree as it is sent out.

If no cache entries exist, the query shipping behaves exactly as in the unmodified Algorithm 7, but sites that have cached previous results must check their caches to see if the query matches any entries.

When a site receives a query, it checks the table fragment timestamps $ts^N_{T_i}$ from the query against entries $c \in \mathcal{C}$ in the local cache to see if any of its cache entries should be invalidated. If $\exists\, T_i, c : ts^N_{T_i} > ts^c_{T_i}$, c is outdated and should be removed. This is done for all cache entries that involve these table fragments, not only those

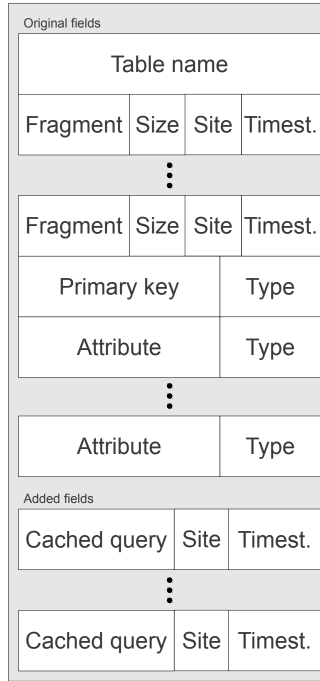


Figure E.3: Extended lookup reply message.

relevant to the current query.

After removing outdated cache entries, the site checks if $\exists\, c\in\mathcal{C} : query(c) = N_n$. If such a cache entry is found, the site has to request current timestamps $ts^*_{T_i}$ of all the table fragments that are referenced by N_n. This is done to guarantee that the cache entry is up-to-date. The locations of the fragments are found by looking at the received query, which contains table scan operators assigned to the corresponding sites.

The cache entry timestamp consists of a set of table fragment timestamps $ts^c_{T_i}$ such that $\forall\, T_i : ts^N_{T_i} \leq ts^c_{T_i} \leq ts^*_{T_i}$. If $\forall\, T_i : ts^c_{T_i} = ts^*_{T_i}$, the site holds back the whole subtree rooted at the cached algebra node and stops query dissemination of that branch. The algebra node is replaced by a special node that delivers the result from cache. If $\exists\, T_i : ts^*_{T_i} > ts^c_{T_i}$, the cache entry is outdated and is removed before query dissemination continues as if the cache entry had never existed, assigning the root node to be processed locally and sending the subtrees rooted at the children of this node to the sites to which they have been assigned by the initiator site.

Figure E.4 shows how the query is distributed to the participating sites. In the example, the query with timestamps is sent out upstream from the initiator site, split at each algebra node. A cache entry for $T * U$ is found on site S_0 and a special request is made to the sites storing relevant table fragments (in this example, the tables consist of only one fragment each) to retrieve the current timestamps. Table T is stored on site S_0, so the timestamp request is processed locally. Site S_1 receives

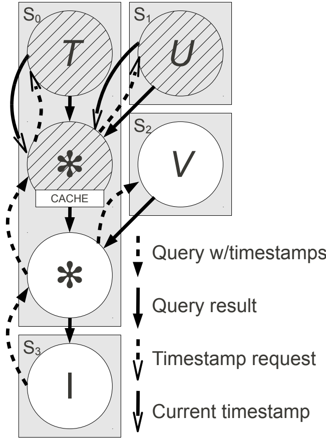


Figure E.4: Query dissemination with table fragment timestamps.

the request from site S_0 and replies with the current timestamp of table U.

Our caching method supports variations of this timestamp policy. Different isolation levels may allow caches that are older than the current table fragment timestamps, leading to more cache hits. Timestamp policies should be the same on all sites, since the system as a whole cannot guarantee stronger isolation levels than the weakest timestamp policy allows.

Dissemination stops when all branches of the query tree have been terminated by a leaf node delivered to the site to which it has been assigned or by a deliver-from-cache operator. As soon as a leaf node is delivered, or a deliver-from-cache node created, the node enters the processing step and starts producing data.

E.4.3 Query Processing and Caching

In the dissemination step, it was discovered whether any relevant cache entries existed and were up-to-date, in which case dissemination of that subtree stopped and the root of the subtree was replaced by a special node serving data from cache. Cache entries are locked in cache as long as they serve an ongoing query.

The special deliver-from-cache operators simply deliver the cached result, including the cached timestamps, which by now are guaranteed to be up-to-date. Since the timestamps of the table fragments needed to produce the cached result are stored with the cache entry, they can easily be retrieved and sent with the result, providing timestamps for caching of downstream nodes.

If data is not served from cache, there might be an opportunity for caching. To allow downstream caches, table fragment timestamps are propagated along with the results of operators, as shown in Figure E.5. At each interior node n in the query tree, the result of the operation is tagged with a combined timestamp $ts_n = \cup_{n' \in children(n)} ts_{n'}$ of the timestamps of all operands.

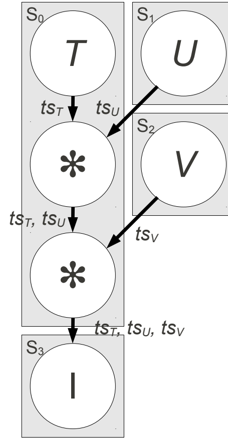


Figure E.5: Result and timestamp propagation.

The result of algebra nodes are candidates for caching at the site where they are processed, and the timestamps that comes with the result are used to timestamp cache entries that are created. The decision to cache the result or not is made when a node starts executing, and is based on the cache replacement policy.

In general, the cache replacement policy defines an ordering of cache entries. In block based caching, where cache entries are always of the same size, only the first entry to be replaced has to be identified. When caching intermediate results of differing sizes, multiple cache entries may have to be removed to fit one larger entry in cache. The ordering must also include queries, so that candidates for caching can be compared against any existing cache entry. In Algorithm 8 we define the general cache replacement algorithm using such an ordering. Given a successor relation $\succ_p$ defined by the cache replacement policy and the cache $\mathcal{C}$, we use the ordered set $\mathcal{C}_p = (\mathcal{C}, \succ_p)$, where $head(\mathcal{C}_p)$ is the least element of $\mathcal{C}_p$.

Each site decides autonomously which results to cache and may apply different cache replacement policies and have different cache sizes. Since cache entries are compared against queries that have not yet been processed, some values must be estimated, e.g., the size of the intermediate result of N_n. The successor relation $\succ_p$ that defines the ordering of cache entries may rely on a number of metrics, either measured or estimated.

Available Metrics

Each site has only a restricted view of the system, knowing only the algebra nodes passing through it and the subtrees rooted at these. It also knows the contents of its own cache and the usage statistics of that cache. We divide the available metrics into three measurement categories: size, cost and query pattern.

The size of the result is necessary to decide if there is room for the result in the cache. However, before the query has been processed and the size can actually

Algorithm 8 Decide to cache the result of N_n (true) or not (false).

At site S, when deciding whether or not to cache the result of N_n:

```
free ← free cache space
C_replaced ← ∅
C_remaining ← 𝒞_p
while free < size(N_n) do
    c ← head(C_remaining)
    if N_n ≻_p c then
        free ← free + size(c)
        C_replaced ← C_replaced ∪ {c}
        C_remaining ← C_remaining \ {c}
    else
        return false
    end if
end while
𝒞_p ← C_remaining
return true
```

be measured, we must rely on estimates based on the available statistics. Table statistics that are available from the global catalog and included in N can be used to find the size of table fragments and estimate the size of the results of downstream operators.

The cost of resolving a query can be estimated knowing operand size and operator implementation details. The cost of reproducing a result from scratch is the cumulative cost of the algebra subtree, so the estimated cost of reproducing the result can be found by adding up the estimated cost of all nodes in the subtree.

A simpler cost estimate is the node's position in the algebra tree. The leaf nodes are table access nodes. Intermediate nodes have a higher cost, and the cost of reproducing the result increases towards the root. Instead of computing the cumulative cost, we can simply use the height of the subtree rooted at that node.

The simplest query pattern metric is to use least recently used (LRU) ordering of cache entries, allowing recently used cache entries to stay in the cache while less recently used entries are replaced.

Cache Replacement Policies

Based on the metrics defined above, we can implement several cache replacement policies.

LRU The simplest cache replacement policy is to always replace the least recently used cache entry. This policy is simple, but is not able to use the semantic information that the caching method makes available. Therefore, it is unable to separate between query results that require a lot of computational effort to reproduce and results that are easily recreated from scratch.

LRU + cost An improvement is to use the cost measure in addition to usage to decide which cache entry should be replaced next. The semantic information allows us to estimate the cost of each step in the query and to use cumulative cost to either favor queries of high or low complexity. The cost measure is given more weight than LRU, but as a cache entry ages also complex results may be replaced. We call these policies LC policies.

The LC$^+$ policy favors results of queries of high complexity, making sure that results that would take longer to reproduce are kept in cache longer than results of queries that have a lower computational complexity. The complex queries are not the most frequently reused, but more time is saved for each cache hit.

Our LC$^-$ policy does the opposite. It adds a penalty to the complex queries. This means that the results of queries of lower complexity, which are expected to be more reusable and produce more cache hits, stays longer in cache. The choice between LC$^+$ and LC$^-$ is a weighing of savings per cache hit versus number of hits.

LRU + height The cost estimate is a quite complex estimate to make. As described earlier, the height of the query tree may be a simple alternative to the full cost estimate. Like cost estimates, height can be used to favor either complex or simple queries, so we divide the LH policies into LH$^+$, which favors results of complex queries, and LH$^-$, which favors versatile results.

E.4.4 Cache Registration

As soon as the last tuple is inserted into a cache entry, it is made available for use. However, it is not registered in the catalog until the site sends an update message to the catalog service.

The sites regularly update the catalog with information about local table fragments, and we extend these updates with information about cache entries. For each entry in the local cache, a site sends a representation of the algebra subtree and timestamps for each table fragment accessed by the subtree.

There are generally more than one table involved in a query. In DASCOSA-DB, the catalog of fragments of one table is stored on one catalog service site. To avoid having to register the cache entries on the catalog service sites for all tables involved in the query, we use a hashing function to select one site to which the query is sent. For each query, the hashing function is used to select the catalog site for one of the involved tables.

Since the cache entry for the result of an algebra tree is always registered at the catalog site storing information on one of the tables referenced by the algebra tree, our method guarantees that by sending the query with the lookup request for all tables, all cache entries are found. Due to the hashing function, the catalog of cache entries is distributed among catalog sites.

Cache Currency and Invalidation

The catalog stores a lower bound on the timestamps of all table fragments. When a catalog site receives an update message for a cache entry c, it can compare the

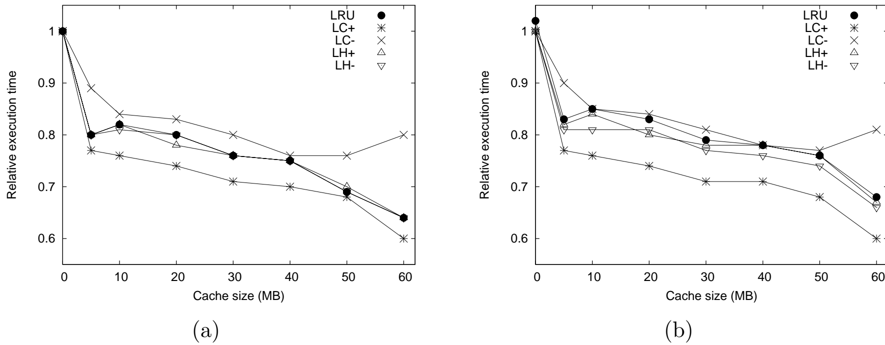


Figure E.6: (a) Relative execution time of high-bandwidth system with uniform workload. (b) Relative execution time of low-bandwidth system with uniform workload.

timestamps $ts^c_{T_i} \in ts_c$ of the cache entry with the timestamps of table fragments $ts^K_{T_i}$ from its part K of the global catalog. If $\exists\, T_i : ts^c_{T_i} < ts^K_{T_i}$, table fragment T_i has been updated after the cache entry was created and this is registered in the catalog. The cache entry will not be registered, and the caching site is informed of the new timestamp.

Similarly, if $\exists\, T_i : ts^c_{T_i} > ts^K_{T_i}$, table fragment T_i has been updated since last catalog update, and $ts^K_{T_i}$ is updated. In this way, the cache entry updates are actually improving the catalog service freshness.

E.4.5 Transactional Support

Implementation of semantic caching does not change transactional support or locking policies in DASCOSA-DB. All locks for a query are acquired by the initiator site before query dissemination, and our semantic caching method does not change that behavior. The initiator site will acquire locks for all table fragments that are needed, including those that form the basis for cache entries used by the query, thereby guaranteeing full transactional isolation.

E.5 Experimental Evaluation

To evaluate the caching method, we have implemented it in the DASCOSA-DB distributed database system prototype. The caching method has been tried with the caching policies described in Section E.4.3 and different query workloads. We used the DASCOSA-DB's existing cost functions for cost estimation.

The experiments were done on a TPC-H [19] dataset using scaling factor $SF = 0.1$, partitioned horizontally based on primary key and distributed over five sites, each running an instance of DASCOSA-DB. One more site with no table fragments was used for issuing queries. We generated query workloads consisting of 200

queries from the TPC-H benchmark queries, varying all substitution parameters of the benchmark. The substitution parameters were drawn either from a uniform distribution or from a skewed distribution where 80% of the values are drawn from 20% of the domain.

We measured the execution time and cache hits of repeated executions of our query workload. These measurements are only meaningful on a relative scale, so execution time was measured relative to a baseline execution without caching. During this execution, caching code was completely disabled. Cache hits were measured relative to the number of queries in the workload.

E.5.1 Varying Network Bandwidth

In this experiment, the network bandwidth was varied to produce two different settings: a high-bandwidth setting with 100 Mbit/s links connecting the sites, and a low-bandwidth setting with 1 Mbit/s links. The substitution parameters of the queries were drawn randomly from the parameter domains, using a uniform distribution.

Figure E.6(a) shows the average execution time relative to the execution time measured when caching was disabled, i.e., without any caching code running. The values shown for 0 MB cache thus show the overhead of the caching method, i.e., the extra cost associated with the caching method without any of its benefits. As is seen from the figure, the overhead is negligible.

Further, the graph shows that with the largest cache size, LRU saves 36% of the execution time, while LC^+ saves 40%. LH^+ and LH^- vary very little from LRU, while LC^-, favoring results of queries of low complexity, performs worst. This indicates that the savings gained from caching results of complex queries outweighs the possibilities for frequently used results of low complexity.

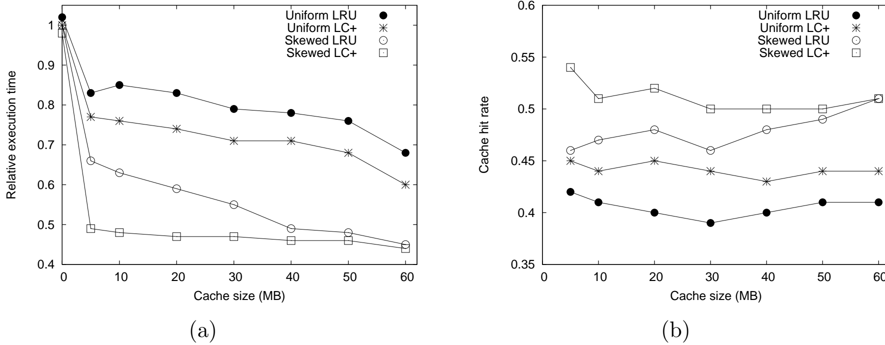
Comparing Figures E.6(a) and E.6(b), we see that the distance between LC^+ and LRU is greater in the low-bandwidth setting. This is in line with our expectations of LC^+, deciding to cache results of more complex queries, allowing for greater savings once they are used. When bandwidth is reduced, LC^+ stands out while the rest of the policies perform poorer.

E.5.2 Varying Parameter Distribution

In this experiment, query parameters were either drawn from a uniform distribution or from a skewed distribution where 80% of the values were drawn from 20% of the parameter domain. Network bandwidth was kept low (1 Mbit/s).

The skewed distribution was chosen to be a more realistic workload, where some values are more frequent than others. This would be the case in many real life systems. We believe the choice of 80/20 is a conservative one, which means slightly pessimistic results.

Figure E.7(a) shows that LRU achieves a reduction in execution time of 55%, i.e., 11 percentage points more than with uniform parameter distribution, and LC^+ saves 56%. While LRU and LC^+ do not differ very much for large cache sizes, LC^+



(a) (b)

Figure E.7: (a) Relative execution time of low-bandwidth system with both workloads. (b) Cache hit rates for low-bandwidth system with both workloads.

achieves these savings also for much smaller cache sizes.

The comparison of cache hit rates in Figure E.7(b) sheds some light on what is going on. LC^+ manages to keep the right entries in cache also for smaller cache sizes, while LRU needs large cache sizes to achieve this. We also note that a larger cache does not necessarily mean better hit rates, since cache entries are of different sizes. Large entries may displace multiple smaller entries, thus reducing the number of hits.

There is clearly an increased hit rate for the skewed workload, which is the reason for the improvement in execution time over the uniform workload.

E.6 Conclusion and Future Work

In this paper, we have developed a new method for semantic caching in a distributed database system with autonomous sites, where caching policies and decisions can vary from site to site and workload statistics are sparse. By making sites autonomous, we allow the system to scale without excessive network traffic. We have shown how the result of subqueries can be cached and reused by subsequent, similar queries to speed up query processing.

We have implemented the semantic cache in the DASCOSA-DB prototype as a proof of concept and platform for our experiments. The experiments have shown that we get considerable improvements in execution time when enabling semantic caching. The overhead of our caching method is also very low.

The cache hit rate is not the only factor influencing the performance. The cost of recomputing the cached data is also important. The savings made possible by caching the result of a complex query are sometimes higher than the savings from caching the results of less time-consuming queries with a higher hit rate. The information necessary to make such decisions is made available by semantic caching.

Our results indicate several ways to further improve query processing by semantic caching. The next step is to further enable the query optimizer to take advantage

of cached intermediate results, including rewriting queries in otherwise suboptimal ways to increase cache hit numbers and rewrite queries to increase reusability of intermediate results. A semantic cache also allows for more advanced cache replacement policies, and further work should be done to find policies that care not only for the number of cache hits, but also the potential computational cost savings of a cache entry.

Bibliography

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of VLDB*, 2003.

[2] H. Andrade, T. M. Kurç, A. Sussman, and J. H. Saltz. Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments. *Parallel Computing*, 33(7-8):497–520, 2007.

[3] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering. Bulletin*, 27(2):11–18, 2004.

[4] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE*, 1995.

[6] B. Chidlovskii, C. Roncancio, and M.-L. Schneider. Semantic cache mechanism for heterogeneous Web querying. *Computer Networks*, 31(11–16):1347–1360, 1999.

[7] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, 1996.

[8] L. M. Haas, D. Kossmann, and I. Ursu. Loading a cache with query results. In *Proceedings of VLDB*, 1999.

[9] J. O. Hauglid, K. Nørvåg, and N. H. Ryeng. Efficient and robust database support for data-intensive applications in dynamic environments. In *Proceedings of ICDE*, 2009.

[10] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of SIGMOD*, 1996.

[11] B. T. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3):302–331, 2006.

[12] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In *Proceedings of SIGMOD*, 2002.

[13] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[14] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000.

[15] D. Lee and W. W. Chu. Semantic caching via query matching for web sources. In *Proceedings of CIKM*, 1999.

[16] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of SIGMOD*, 2002.

[17] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Record*, 30(2):307–318, 2001.

[18] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of MobiCom*, 2000.

[19] Transaction Processing Performance Council. TPC benchmark H (decision support) standard specification revision 2.11.0, 2010.

Paper F

Efficient Distributed Top-k Query Processing with Caching

Norvald H. Ryeng, Akrivi Vlachou, Christos Doulkeridis and Kjetil Nørvåg.
In *Proceedings of DASFAA*, 2011.

Abstract

Recently, there has been an increased interest in incorporating in database management systems rank-aware query operators, such as top-k queries, that allow users to retrieve only the most interesting data objects. In this paper, we propose a cache-based approach for efficiently supporting top-k queries in distributed database management systems. In large distributed systems, the query performance depends mainly on the network cost, measured as the number of tuples transmitted over the network. Ideally, only the k tuples that belong to the query result set should be transmitted. Nevertheless, a server cannot decide based only on its local data which tuples belong to the result set. Therefore, in this paper, we use caching of previous results to reduce the number of tuples that must be fetched over the network. To this end, our approach always delivers as many tuples as possible from cache and constructs a remainder query to fetch the remaining tuples. This is different from the existing distributed approaches that need to re-execute the entire top-k query when the cached entries are not sufficient to provide the result set. We demonstrate the feasibility and efficiency of our approach through implementation in a distributed database management system.

F.1 Introduction

Nowadays, due to the huge amount of available data, users are often overwhelmed by the variety of relevant data. Therefore, database management systems offer rank-aware query operators, such as top-k queries, that allow users to retrieve only a limited set of the most interesting tuples. Top-k queries [5, 8, 15] retrieve the k tuples that best match the individual user preferences based on a user-specified scoring function. Different scoring functions express the preferences of different users. Several applications benefit from top-k queries, including web search, digital libraries and e-commerce. Moreover, the high distribution of data raises the importance of supporting efficient top-k query processing in distributed systems.

In this paper, we propose a cache-based approach, called *ARTO*[1], for efficiently supporting top-k queries in distributed database management systems. In large-scale distributed systems, the dominant factor in the performance of query processing is the communication cost, measured as the number of tuples transmitted over the network. Ideally, only the k tuples that belong to the result set should be fetched. Nevertheless, in the case of top-k queries, a server cannot individually decide which of its top-k local tuples belong to the global top-k result set of the query. In order to restrict the number of fetched tuples and reduce the communication costs, we employ caching of result sets of previously posed top-k queries. Each server autonomously maintains its own cache and only a summary description of the cache is available to any other server in the network.

In general, a top-k query is defined by a scoring function f and a desired number of results k, and these parameters differ between queries. Given a set of cached

[1] Algorithm with Remainder TOp-k queries.

top-k queries in the system and a new query, the problem is to identify whether the results of cached queries are sufficient to answer the new query. To deal with this problem, we apply techniques similar to those of the view selection problem in the case of materialized views [15] in centralized database systems. Based on the cached queries, we need to decide whether the cached results *cover* the results of a new query. In this case, the query is answered from the cache and no tuples need to be transferred over the network. However, the major challenge arises when the query is not covered by the cached tuples.

Different from existing approaches [20] that require the servers to recompute the query from scratch, we do not evaluate the entire query, but we create a *remainder query* that provides the result tuples that are not found in cache. More detailed, we split the top-k query into a top-k' query ($k' < k$) that is answerable from cache, and a remainder next-($k - k'$) query that provides the remaining tuples that were not retrieved from the top-k' query. To further optimize the query performance, we deliberately assign the top-k query to the server that is expected to induce the lowest network cost based on the locally cached tuples. To summarize, the contributions of this paper are:

- We propose a novel framework for distributed top-k queries that retrieves as many tuples k' ($k' < k$) as possible from the cache, and poses a *remainder query* that provides the remaining $k - k'$ tuples that are not found in cache.
- We present a novel method for efficiently computing remainder queries, without recomputing the entire top-k query.
- We propose a server selection mechanism that identifies the server that owns the cache with the most relevant entries for a given query.
- We evaluate our approach experimentally by integrating ARTO in an existing distributed database management system [14], and we show that our method significantly reduces communication costs.

The rest of this paper is organized as follows. In Section F.2, we explain how this paper relates to previous work in this area. Section F.3 presents preliminary concepts, and Section F.4 presents our framework for distributed top-k query processing. Answering top-k queries from cache is outlined in Section F.5. The remainder queries are described in Section F.6, while Section F.7 presents the server selection mechanism. Results from our experimental evaluation are presented in Section F.8, and in Section F.9 we conclude the paper.

F.2 Related Work

Centralized processing of top-k queries has received considerable attention recently [2, 5, 8, 15]. For a comprehensive survey of top-k query processing we refer to [16]. Hristidis et al. [15] discuss how to answer top-k queries from a set of materialized ranked views of a relational table. Each view stores all tuples of the relation ranked according to different ranking functions. The idea is to materialize a set of views based on

a requirement either on the maximum number of tuples that must be accessed to answer a query, or on the maximum number of views that may be created. When a query arrives, one of these views is selected to be used for answering the top-k query. In [11], the materialized views of previous top-k queries (not entire relations) are used to answer queries, as long as they contain enough tuples to satisfy the new query. For each incoming query, the view selection algorithm chooses a set of views that will give an optimal (in terms of cost) execution of the proposed LPTA algorithm. A theoretical background on view selection is given in [3], providing theoretical guarantees whether a view is able to answer a query or not. However, the algorithms that are presented only allow a query to be answered from views if the views are guaranteed to provide the answer.

In distributed top-k query processing, the proposed approaches can be categorized based on their operation on vertically [6, 9, 12, 17, 18] or horizontally [1, 4, 19, 20] distributed data. In the case of vertically distributed data, any server maintains only a subset of the attributes of the complete relation. Then, each server is able to deliver tuples ranked according to any scoring function that is applied on one or more of its attributes [9, 12, 17]. The TPUT algorithm [6] focuses on limiting the number of communication round-trips, and this work has later been improved by KLEE [18].

In the case of horizontally distributed data, each server stores a subset of the tuples of the complete relation, but for each tuple all attributes are maintained. In [1], a broadcasting technique for answering top-k queries in unstructured peer-to-peer networks is presented. For super-peer topologies, Balke et al. [4] provides a method using indexing to reduce communication costs. This method requires all super-peers to process queries, unless exactly the same query reappears. SPEERTO [19] precomputes and distributes skyline result sets of super-peers in order to contact only those super-peers that are necessary at query time. BRANCA [20] is a distributed system for answering top-k queries. Caching of previous intermediate and final results is used to avoid recomputing parts of the query. The cache is used much in the same way as the materialized views in [3, 11, 15], but on intermediate results of the query. This means that some servers in the system must process the query from scratch, while others may answer their part of the same query from cache. The main difference between ARTO and other caching approaches, such as BRANCA, becomes clear in the hard cases, when the query cannot be answered by the cache. ARTO still uses the part of the cache that partially answers the query and poses a remainder query for the remaining tuples, without the need to process the query from scratch, as in the case of BRANCA.

Finally, our techniques for answering top-k queries relate to stop-restart of query processing [7, 10, 13]. These methods assume that some of the result tuples are already produced and restart processing from where the original query stopped. Our remainder queries differ by not restarting an existing top-k query but a query that was partially answered by cached tuples.

F.3 Preliminaries

Top-k queries are defined based on a monotone function f that combines the individual scores into an overall scoring value, that in turn enables the ranking (ordering) of tuples. Given a relation R, which consists of n attributes a_i, the result set of a top-k query $Q = \langle R, f, k \rangle$ contains k tuples such that there exists no other tuple in R with better score than the k tuples in the result set. The relation R may be a base relation or the result of an algebra operator, i.e., the result of a join. The most commonly used scoring function is the weighted sum function, also called linear. Each attribute a_i is associated with query-dependent weight w_i indicating a_i's relative importance for the query. Furthermore, without loss of generality, we assume that for any tuple t and any attribute a_i the values $t(a_i)$ are scaled to $[0, 1]$. The aggregated score $f(t)$ for a tuple t is defined as a weighted sum of the individual scores: $f(t) = \sum_{i=1}^{n} w_i t(a_i)$, where $w_i \geq 0$ $(1 \leq i \leq n)$, and $\exists j$ such that $w_j > 0$. The weights represent the relative importance of different attributes, and without loss of generality we assume that $\sum_{i=1}^{n} w_i = 1$. Thus, a linear top-k query Q is defined by a vector $\mathbf{w_Q}$ and the parameter k. The ranked tuples can be delivered in either ascending or descending order, but for simplicity, we will only consider descending order in this paper. Our results are also valid in the ascending case.

A tuple t of R can be represented as a point in the n-dimensional Euclidean space. Furthermore, given a top-k query $Q = \langle R, f, k \rangle$ defined by a linear scoring function, there exists a one-to-one correspondence between the weighting vector $\mathbf{w_Q}$ and the hyperplane which is perpendicular to $\mathbf{w_Q}$. We refer to the (n-1)-dimensional hyperplane, which is perpendicular to vector $\mathbf{w_Q}$ and crosses the k-th result tuple, as the *query plane* of $\mathbf{w_Q}$, and denote it as L_Q. All points on the query plane L_Q have the same scoring value for $\mathbf{w_Q}$. A 2-dimensional example is depicted in Fig. F.1. Processing the top-k query Q is equivalent to sweeping the line L_Q from the upper right corner towards the lower left corner. Each time L_Q meets a tuple t, this tuple is reported as the next result tuple. When L_Q meets the k-th tuple, the complete result set has been retrieved.

F.4 ARTO Framework

In this paper, we assume a distributed database system where the relations are horizontally fragmented over multiple servers. In more details, each relation R is fragmented into a set of fragments $R_1, R_2, \ldots, R_f$ and each fragment R_i consists of a subset of tuples of the relation R. Our approach is generic and imposes no further constraints on the way fragments are created or whether they are overlapping or not. Furthermore, each server may store fragments of different relations. Any server can pose a top-k query and we refer to that server as *querying server*. During query processing, the querying server may connect to any other server. Thus, no routing paths are imposed on the system other than those of the physical network itself. The only assumption of ARTO is that there exists a distributed *catalog* accessible to all servers, which indexes the information about which server stores fragments of each relation R. Such a distributed catalog can be implemented using a distributed

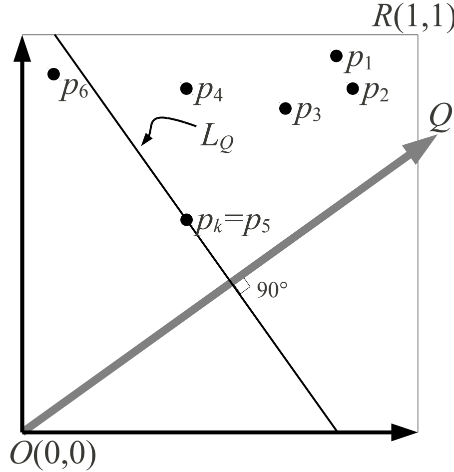


Figure F.1: 2D representation of query and data space.

hash table (DHT).

To answer a top-k query over a relation R, the querying server first locates those servers that store fragments of R by using the catalog, and constructs a query plan such as the one in Fig. F.2(a). In our example, S_2 is the querying server and the relation R is fragmented in four fragments $R_1, \ldots, R_4$ stored on servers $S_1, \ldots, S_4$ respectively. Based on the query plan, each fragment R_i is scanned in ranked order (denoted in Fig. F.2(a) as rank), and the top-k operator reads tuples one by one, until the k highest scoring tuples have been retrieved. In more details, the top-k operator maintains a sorted output queue and additionally a list containing the score of the last tuple from each server. Since the tuples read from R_i are in ranked order, whenever a tuple in the output queue has a higher score than all scores in the list, it can safely be output as a result tuple. Thus, the top-k tuples are returned incrementally. Moreover, the top-k operator reads the next tuple from the fragment R_i with the tuple with the highest score in the list. Therefore, the top-k operator reads as few input tuples as possible from the fragments R_i.

This is the basic approach of answering top-k queries in a distributed data management system. Since it is important to minimize the network cost of query processing, ARTO uses a caching mechanism to take advantage of previously answered top-k queries. Thus, ARTO avoids retrieving tuples from other servers, when the cached tuples are sufficient to answer the new query. To this end, each server maintains its own cache locally, and caches the queries (and their results sets) that were processed by itself. During query processing, the querying server first uses its cache to detect whether the cached tuples are sufficient to answer the given top-k query (see Section F.5). Even if the cached tuples are not sufficient, ARTO minimizes the transferred data by using as many cached tuples as possible and retrieving only the missing tuples from the remote servers through the novel use of *remainder queries* (see Section F.6). To this end, the query plan is rewritten in order to take advantage

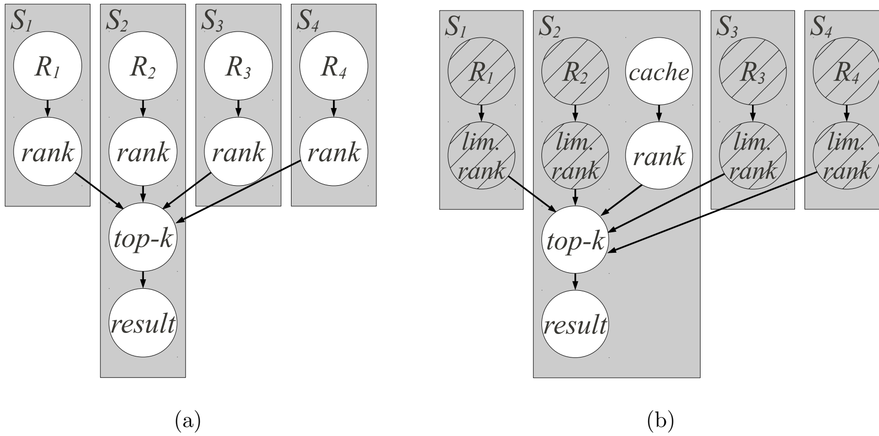


Figure F.2: (a) Query plan for distributed top-k query. (b) Transformed query plan.

of the local cache. The result of such a query transformation is shown in Fig. F.2(b). Compared to the initial query plan, the top-k operator additionally retrieves tuples from the cache and performs a limited scan from the relation fragments, thus transferring only tuples that are not cached.

The combination of cached tuples and remainder queries allows ARTO to reduce the number of transferred tuples. The exact number of transferred tuples depends on the similarity of cached queries to the new query. Thus, in order to improve further the query processing performance, we extend ARTO with a *server selection* mechanism, which assigns the new query to the server with the most similar cached query. In order to facilitate this mechanism, each server publishes descriptions of its cached queries in the distributed catalog. Then, the querying server first detects the server with the most similar cached query, and re-assigns the new query to this server (see Section F.7).

In rest of this paper, we assume that data tuples are not updated, inserted or deleted during query processing. This means that the cache always will be up-to-date. Techniques that enforce cache consistency can be adopted in a straightforward way, as they are orthogonal to our work.

F.5 Answering Top-k Queries from Cache

In ARTO, each server autonomously maintains its own cache. More specifically, after answering a top-k query and retrieving the entire result set, the query originator is able to cache the query result. The cache $\mathcal{C} = \{C_i\}$ maintains a set of m *cache entries* C_i. Each cache entry $C_i = \{Q_i, b_i, \{p_1, \ldots, p_{k_i}\}\}$ is defined by a query $Q_i = \{R, f_i, k_i\}$, the tuples $\{p_1, \ldots, p_{k_i}\}$ that belong to the result set of Q_i, and a *threshold* b_i which is the scoring value of point p_{k_i} with respect to f_i, i.e., $b_i = f_i(p_{k_i})$. Consequently, any tuple p of the cache entry C_i has score $f_i(p) \geq b_i$. Notice that

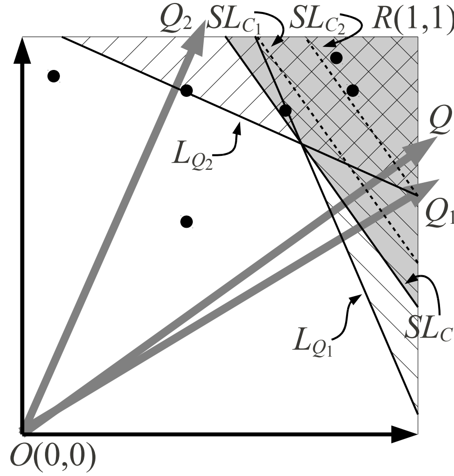


Figure F.3: Cache containing the cache entries of two queries.

the description of a cached entry C_i that is published in the catalog consists only of $\{Q_i, b_i\}$, without the individual result tuples. For the sake of simplicity, we assume that all cache entries refer to the same relation R. Obviously, given a query $Q = \{R, f, k\}$, only cache entries that refer to relation R are taken into account for answering Q.

Fig. F.3 shows a server's cache $\mathcal{C}$ that contains two cache entries, C_1 and C_2. Query Q_1 corresponds to a top-3 query, while Q_2 is a top-4 query with different weights. Their corresponding lines, L_{Q_1} and L_{Q_2}, stop at the k-th point for each query respectively.

F.5.1 Basic Properties

In this section, we analyze when the query results of a cache $\mathcal{C}$ are sufficient to answer a top-k query Q. When this situation occurs, we say that the cache *covers* the query. Given a query $Q = \{R, f, k\}$, we identify three cases of covering: (1) a cache entry C_i covers a query defined by the same function ($f = f_i$), (2) a cache entry C_i covers a query defined by a different function ($f \neq f_i$), and (3) a set of cache entries $\{C_i\}$ cover a query defined by a different function ($f \neq f_i$, $\forall i$).

In the first case, if there exists a cache entry C_i such that the weighting vectors that define f and f_i are identical and $k \leq k_i$, then Q can be answered from the result of the cache entry C_i. More specifically, the first k data points of the cache entry C_i provide the answer to Q.

In the second case, we examine if a cache entry covers a query defined by a different function. To this end, we use the concept of *safe area* [3] SA_i of a cache entry C_i.

Definition 1 ***(Safe area)*** *The safe area SA_i of a cache entry C_i with respect to a query Q is the area defined by the right upper corner of the data space and*

the $(n-1)$-dimensional hyperplane SL_{C_i} that is perpendicular to the query vector, intersects the query plane L_{Q_i}, and has the largest scoring value for Q between all candidate hyperplanes.

In Fig. F.3, the lines that define the safe areas for C_1 and C_2 with respect to Q are shown as SL_{C_1} and SL_{C_2}, respectively. Given a query Q, a cache entry C_i is sufficient to answer a query Q, if it holds that the safe area SA_i of the cache entry C_i contains at least k data points. This means that there cannot be any other tuples in the result set of Q that have not been retrieved by the query Q_i, because the safe area has been scanned during the processing of Q_i. For example, in Fig. F.3, both cache entries are sufficient for answering the query Q for $k = 2$, but none of those is sufficient to answer the query Q for $k = 3$.

The third case deals effectively with the previous situation. Several cache entries need to be combined to answer the top-k query, since a single cache entry is not sufficient. To determine whether a set of cache entries can be used to answer a top-k query, we define the concept of *cache horizon*.

Definition 2 ***(Cache horizon)*** *The cache horizon of a cache $\mathcal{C} = \{C_i\}$ is defined as the borderline of the area defined by the union of query planes L_{Q_i}.*

The cache horizon represents the border between the points that are cached and those that are not. Points behind the cache horizon (towards the right upper corner of the data space) are contained in at least one cached entry, while points beyond the cache horizon (near the origin of the data space) have to be retrieved from the relation R that is stored at different servers. In Fig. F.3, the cache horizon is defined by the lines L_{Q_1} and L_{Q_2} and the enclosed area has been examined to answer queries Q_1 and Q_2. In order to determine if the result set of Q is behind the cache horizon and can be answered by combining more than one cache entry, we define the *limiting point* of the cache.

Definition 3 ***(Limiting point)*** *The limiting point of a cache $\mathcal{C}$ is the point, where the hyperplane $SL_{\mathcal{C}}$ perpendicular to the query vector intersects the cache horizon, when $SL_{\mathcal{C}}$ moves from the right upper corner of the data space towards the origin.*

The area defined by the hyperplane $SL_{\mathcal{C}}$ and the right upper corner of the data space is called *safe area of the cache horizon*. If this area contains more than k data points, then Q can be answered by combining more than one cache entry.

Given a cache $\mathcal{C}$ with m cache entries $C_1, C_2, \dots, C_m$, the limiting point of the horizon with respect to a query Q can be identified using linear programming. We construct a constraint matrix $\mathbf{H}$ and right-hand-side values $\mathbf{b}$ from the weights and thresholds of the m cache entries:

$$\mathbf{H} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & & & \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Given a query Q, the objective is to maximize $f = \mathbf{w_Q}^{\mathrm{T}}\mathbf{a}$, subject to the constraints $\mathbf{Ha} \leq \mathbf{b}$ and $0 < a_i < 1$, $\forall a_i$. The solution of this linear programming problem provides the coordinates of the limiting point. By applying the scoring function f defined by Q, we get the *cache score* $b = f(p)$ of the limiting point p. If at least k cached points p_i exist such that $f(p_i) \geq b$, then the entire result set of query Q is retrieved from the cache entries.

F.5.2 Cache Replacement Policy

A first observation regarding a cache entry $C_i \in \mathcal{C}$ is that it can become *redundant* due to other cache entries in $\mathcal{C}$. More formally, C_i becomes redundant if its query Q_i is covered by a set of other cache entries. Redundant cache entries can be evicted from the cache without affecting the cache's ability to answer queries. Identifying whether a cache entry C_i is redundant is achieved by solving a linear programming problem. More detailed, the objective is to maximize $\mathbf{w_{C_i}}^{\mathrm{T}}\mathbf{a}$, subject to $\mathbf{H'a} \leq \mathbf{b'}$ and $\forall a_j : 0 < a_j < 1$, where $\mathbf{H'}$ and $\mathbf{b'}$ describe the combined horizon of all cache entries except C_i. Thus, we find the limiting point p of the cache when C_i is ignored. If $b_i > f_i(p)$, the cache entry C_i is redundant and can be removed.

Applying a traditional cache replacement policy, such as LRU, is inappropriate due to the unique characteristics of our cache. The reason is that answering a top-k query from the cache may require combining tuples from more than one cache entry. Consequently, cache entries are utilized collectively, rendering any policy based on usage statistics of individual cache entries ineffective.

Motivated by this, we introduce a new cache replacement policy named *Least-Deviation Angle* (LDA), which is particularly tailored to our problem. After removing redundant entries, LDA determines the priority of a cache entry to be evicted based on deviation from the equal-weights vector $\mathbf{e}^{\mathrm{T}} = (1, 1, \ldots, 1)$. For each cache entry C_i, the angle $\theta_i = \arccos(\hat{\mathbf{w_{C_i}}} \cdot \hat{\mathbf{e}})$ between $\mathbf{e}$ and C_i is calculated and used as a measure of deviation. The entry C_i with the largest θ_i is replaced first. Intuitively, LDA penalizes cache entries that have low probability to be highly similar to other queries.

F.6 Remainder Queries

In the previous section, we described in which cases the entries of the cache are sufficient for retrieving the entire result set of a query Q. When this occurs, no networking cost exists for answering the query. In the case where only $k' < k$ tuples t are retrieved from the cache for which the inequality $f(t) \geq b$ holds (b is the cache score), the cache fails to return the complete result set. Then, instead of executing the entire query Q from scratch, ARTO executes a *remainder query* that retrieves only the $k - k'$ missing tuples and transfers only the necessary tuples to provide the complete result set. We first provide a short discussion showing that is more beneficial to execute a remainder query, rather than restarting a cached query $Q_i = \{R, f_i, k_i\}$ and retrieving additional tuples, so that the k tuples of Q are

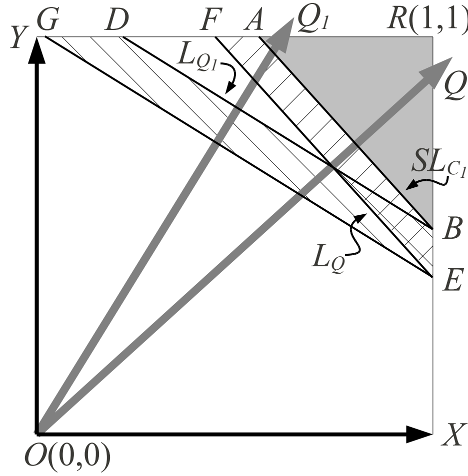


Figure F.4: Areas examined by the remainder query vs. restarting a query Q_1.

retrieved. Then, we define the remainder query and explain how it will be processed in order to minimize the network consumption.

F.6.1 Discussion

In this section, we discuss the issue whether it is more beneficial to restart a query of a cache entry C_i than posing a remainder query. Fig. F.4 depicts a cache containing one cache entry C_1 that covers the data space until the line L_{Q_1} (line DB). A query Q is posed, and the points in the cache entry until the line SL_{C_1} (line AB) are used for answering the query Q. If fewer than k tuples are enclosed in ABR, additional uncached tuples must be retrieved from remote servers. We consider two options for retrieving the remaining tuples. The first alternative is to pose a remainder query that would scan the part $FEBA$ of the data space. Since the query is executed based on the given weighting vector of Q, we can stop after retrieving k tuples exactly, i.e., at the query line L_Q (FE). The other alternative is to restart the cached query Q_1. In this case, we can take advantage of all k_1 data points of the cache entry C_1 (i.e., we save the cost of scanning DBA). On the other hand, in order to be sure that we have retrieved all tuples of the result set of Q we have to scan a larger area at least until the line GE.

If data is uniformly distributed, the number of tuples retrieved is proportional to the area of the data space that is scanned. For the sake of simplicity, we assume that the query line of any query lies in the the upper right triangle of the data space. This means that we have scanned less than half the data space, in order to retrieve the result set of any query, which is an acceptable assumption since usually the values of k are small. In our example, the area of $FEBA$ is smaller than the area of $GEBD$, and the retrieved tuples are expected to be fewer when the remainder query is used. In the following, we prove that this always holds for the 2-dimensional case,

when the query line does not cross the diagonal line XY. Similar conclusions can been drawn for arbitrary dimensionality.

Theorem 1 *Given a 2-dimensional data space, if all query lines do not cross the diagonal line XY, a smaller area is scanned if the remainder query is executed than if continuing a cached query.*

Proof 1 *Using the areas of Fig. F.4, it suffices to show that the area of trapezoid $FEBA$ is smaller than the area of trapezoid $GEBD$. The two trapezoids share one common side, namely EB. Furthermore, it is always the case that $BD > BA$ and $GE > FE$. Based on Thales' theorem about the ratios of line segments that are created if two intersecting lines are intercepted by a pair of parallels, we derive that $\frac{FA}{AR} = \frac{EB}{BR}$ (1) and $\frac{GD}{DR} = \frac{EB}{BR}$ (2). From (1) and (2) we conclude that $\frac{FA}{AR} = \frac{GD}{DR}$. Since $DR > AR$, we derive that $GD > FA$. Therefore, three sides of $FEBA$ are smaller than the corresponding three sides of $GEBD$ and the remaining fourth side BE is common. Hence, the area of $FEBA$ is smaller than the area of $GEBD$.*

F.6.2 Processing of Remainder Queries

Given a query Q and a cache score b, a remainder query is defined as $Q' = \langle R, f, k - k', b\rangle$, where k' is the number of cached tuples p such that $f(p) \geq b$. Any server S_i that stores a fragment R_i of the relation R receives the remainder query Q'. Independently from the implementation of the top-k operator at S_i, the server S_i transfers to the querying server only tuples p such that $f(p) \leq b$. Thus, it avoids transferring tuples that are already cached and lie in the safe area of the querying server.

To further limit the number of transferred tuples to the querying server, S_i filters out some of the locally retrieved tuples by using the cache horizon before transferring them. Even though some tuples lie outside the safe area, they are available at the querying server in some cache entry. For example, in Fig. F.4, the remainder query has to start scanning the data space from the line SL_{C_1} until k tuples are retrieved, i.e., the remainder query fetches new tuples until the query line L_Q. Nevertheless, the points that fall in the triangle DBA are already available at the querying server in the cache entry C_1. These tuples do not need to be transferred, thus minimizing the number of transferred data. In order for S_i to be able to filter out tuples based on the cache horizon, S_i retrieves the descriptions of all cache entries from the querying server. Then, all tuples p such that there exists a cache entry C_i such that $f_i(p) > b_i$ are not transferred to the querying server, since these tuples are stored locally in the cache. The querying server combines the tuples received from the servers S_i with the tuples in the cache and produces the final result set of the query Q. To summarize, the cache horizon is used to limit the remainder query, which means that the whole cache is exploited and a minimal number of tuples is fetched from other servers.

Algorithm 9 Server selection

1: **Input:** Query $Q = \{R, f, k\}$, Servers $\mathcal{S}$
2: **Output:** Server S^* that will process Q
3: $S^* \leftarrow null$, $minScore \leftarrow \infty$
4: **for** $(\forall S_i \in \mathcal{S})$ **do**
5: $\{(Q_j, b_j)\} \leftarrow catalog$.getCacheDesc($S_i$)
6: $score(S_i) \leftarrow$ computeLimitingPoint($\{(Q_j, b_j)\}$)
7: **if** $(score(S_i) < minScore)$ **then**
8: $S^* \leftarrow S_i$
9: $minScore \leftarrow score(S_i)$
10: **end if**
11: **end for**
12: **return** S^*

F.7 Server Selection

The problem of server selection is to identify the best server for executing the top-k operator. While the rank scan operators must be located at the servers that store the relation fragments, the top-k operator can be placed on any server. Our server selection algorithm assigns the top-k operator to the server that results in the most cost-efficient query execution in terms of network cost.

Intuitively, the best server S^* to process the top-k query $Q = \{R, f, k\}$ is the one that can return as many as possible from the k result tuples from its local cache, thereby reducing the amount of the remaining result tuples that need to be fetched. To identify S^*, we need to inspect the cache entries for each server. This operation is efficiently performed using the distributed catalog. In more detail, the catalog can report the descriptions of cache entries $\mathcal{C} = \{C_i\}$ of any server, where a description of C_i consists of $\{Q_i, b_i\}$. Based on this information, the limiting point of the server is calculated, as described in Section F.5. Based on the limiting point, we compute the score of each server by applying the function f of the query Q. The server S^* with the smallest score is selected because this server has the largest safe area and therefore is the best candidate to process the top-k query. Algorithm 9 provides the pseudocode for the server selection mechanism.

F.8 Experiments

In this section, we present an experimental evaluation of ARTO. We have implemented ARTO into the DASCOSA-DB [14] distributed database management system and use this implementation to investigate the effect of different parameters, query workloads and datasets.

Experimental setup. DASCOSA-DB provides a global distributed catalog based on a distributed hash table, and this catalog was used to implement publishing and lookup of cache entries' descriptions. Experiments were performed for varying a) number of servers, b) values of k, and c) cache size. We used three datasets,

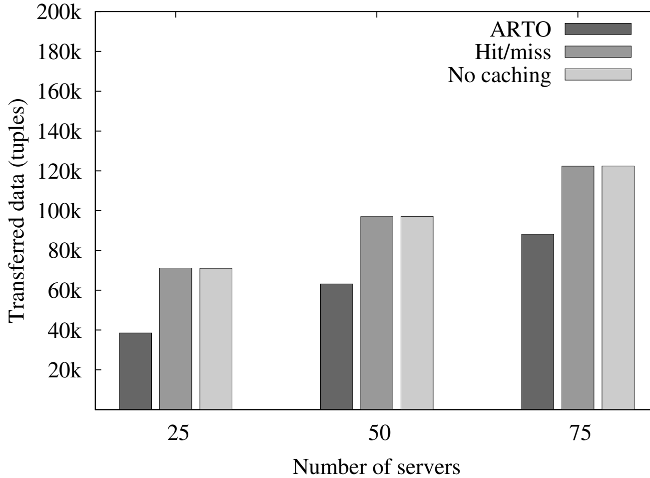


Figure F.5: Transferred data for 1,000 queries and uniform data distribution.

with uniform, correlated and anti-correlated distributions. Each dataset consisted of 1,000,000 5-dimensional tuples. The datasets were distributed horizontally and uniformly over all servers. A separate querying server issued queries to the system and received the results. The weights of the queries were uniformly distributed.

Each experiment was performed both without caching and with ARTO enabled. In addition, we did experiments with a hit/miss implementation where the cache was used only if it were sufficient to answer the complete query. This is conceptually similar to previously proposed methods, e.g., BRANCA [20]. We measured the number of tuples accessed a) locally on the querying server using its cache, and b) from remote servers.

Varying number of servers. In this experiment, ARTO was evaluated with uniformly distributed, correlated and anti-correlated datasets. Each dataset was distributed over 25, 50 and 75 servers. A workload of 1,000 queries with uniformly distributed weights and $k = 50$ were issued. Each server had 10,000 bytes cache, which allows for 4 complete top-50 results to be cached at each server.

Fig. F.5 shows the total number of tuples that are transferred over the network for the query workload using a uniform dataset. We observed similar results for correlated and anti-correlated datasets, which hints that the performance of our approach is stable across different data distributions. The combination of server selection with remainder queries causes a major improvement in network communication costs, even with such a small cache size (4 cache entries). The advantage of ARTO is clearly demonstrated when comparing to the hit/miss strategy, which performs poorly, as it requires k tuples in the safe area to use the cache. Since cache misses are frequent, the complete top-k query has to be executed. The results of hit/miss are just barely better than without caching, while ARTO achieves significant improvements.

Varying *k*. The size of k affects the gain that can be obtained from caching. If k

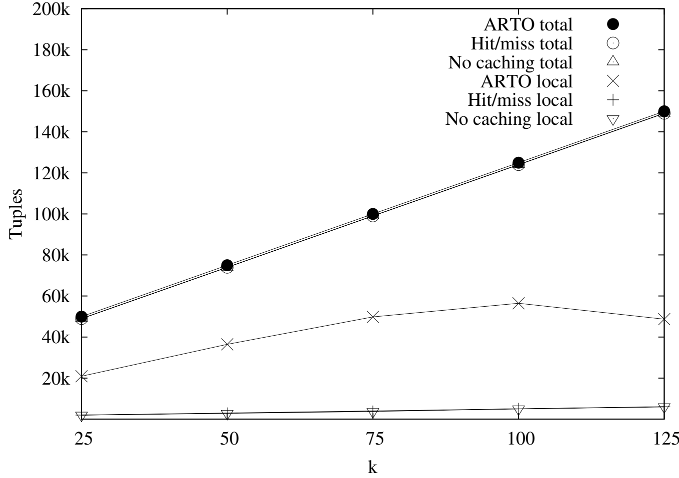


Figure F.6: Results of queries with varying k.

is very small, there are not that many remote accesses that can be replaced by local accesses. In this experiment, the caching method was tested with varying values for k. A uniform dataset of 1,000,000 tuples on 25 servers was used. Each site had 10,000 bytes cache. The results displayed in Fig. F.6 show how the number of total and local accesses increases with increasing k. ARTO always accesses significantly more local tuples compared to the competitor approaches. Around $k = 100$, the number of local tuples accessed starts to decrease. This is because the cache is of a limited size. With $k = 100$, only two complete top-k results fit in cache. Even in this extreme case, ARTO still manages to access a high percentage of the total tuples from the local cache, thereby saving communication costs.

Cache size. In order to study the effect of cache size in more detail, we performed an experiment where we gradually increased the cache size up to 50,000 bytes, i.e., more than 20 complete results. We fixed $k = 50$ and used a uniform dataset of 1,000,000 tuples on 25 servers. The results are shown in Fig. F.7. As the cache size increases, more top-k queries can be cached, thus enlarging the safe area. Consequently, ARTO reduces the number of transferred data (remote tuples accessed). In contrast, the hit/miss strategy always results in cache misses and cannot reduce the amount of transferred data.

F.9 Conclusion

In this paper, we present ARTO, a novel framework for efficient distributed top-k query processing. ARTO relies on a caching mechanism that reduces the network communication costs significantly by retrieving as many tuples as possible from the local cache. In order to retrieve the missing tuples, we define the remainder query that transfers only the tuples that are not stored in the cache by filtering out tuples based on the cache horizon. Moreover, ARTO provides a server selection mechanism

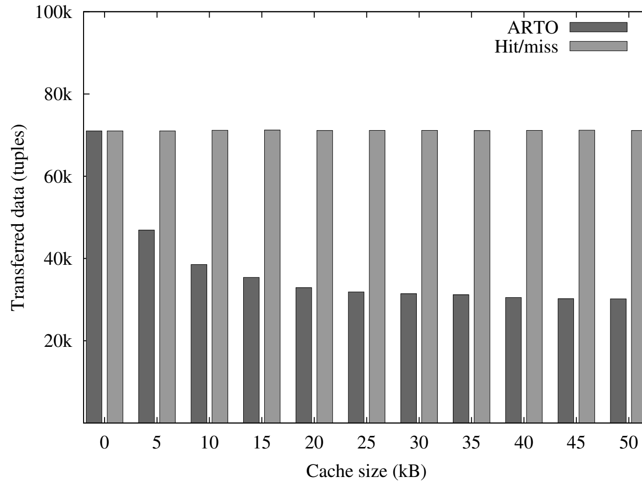


Figure F.7: Results of queries with varying cache size.

that assigns a new top-k query to the most promising server. We have implemented our framework in the DASCOSA-DB database management system. The results of the experiments show considerable improvements in network communication costs.

Acknowledgments

The authors would like to express their thanks to Jon Olav Hauglid for help with the implementation in DASCOSA-DB and João B. Rocha-Junior for providing the dataset generator.

Bibliography

[1] Akbarinia, R., Pacitti, E., Valduriez, P.: Reducing network traffic in unstructured P2P systems using top-k queries. Distributed and Parallel Databases 19(2-3), 67–86 (2006)

[2] Akbarinia, R., Pacitti, E., Valduriez, P.: Best position algorithms for top-k queries. In: Proceedings of VLDB (2007)

[3] Baikousi, E., Vassiliadis, P.: View usability and safety for the answering of top-k queries via materialized views. In: Proceedings of DOLAP (2009)

[4] Balke, W.T., Nejdl, W., Siberski, W., Thaden, U.: Progressive distributed top-k retrieval in peer-to-peer networks. In: Proceedings of ICDE (2005)

[5] Bruno, N., Chaudhuri, S., Gravano, L.: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM Trans. Database Syst. 27(2), 153–187 (2002)

[6] Cao, P., Wang, Z.: Efficient top-k query calculation in distributed networks. In: Proceedings of PODC (2004)

[7] Chandramouli, B., Bond, C.N., Babu, S., Yang, J.: Query suspend and resume. In: Proceedings of SIGMOD (2007)

[8] Chaudhuri, S., Gravano, L.: Evaluating top-k selection queries. In: Proceedings of VLDB (1999)

[9] Chaudhuri, S., Gravano, L., Marian, A.: Optimizing top-k selection queries over multimedia repositories. IEEE Trans. on Knowledge and Data Engineering 16(8), 992–1009 (2004)

[10] Chaudhuri, S., Kaushik, R., Ramamurthy, R., Pol, A.: Stop-and-restart style execution for long running decision support queries. In: Proceedings of VLDB (2007)

[11] Das, G., Gunopulos, D., Koudas, N., Tsirogiannis, D.: Answering top-k queries using views. In: Proceedings of VLDB (2006)

[12] Güntzer, U., Balke, W.T., Kießling, W.: Optimizing multi-feature queries for image databases. In: Proceedings of VLDB (2000)

[13] Hauglid, J.O., Nørvåg, K.: PROQID: Partial restarts of queries in distributed databases. In: Proceedings of CIKM (2008)

[14] Hauglid, J.O., Nørvåg, K., Ryeng, N.H.: Efficient and robust database support for data-intensive applications in dynamic environments. In: Proceedings of ICDE (2009)

[15] Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER: A system for the efficient execution of multi-parametric ranked queries. In: Proceedings of SIGMOD (2001)

[16] Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 40(4) (2008)

[17] Marian, A., Bruno, N., Gravano, L.: Evaluating top-k queries over web-accessible databases. ACM Trans. Database Syst. 29(2), 319–362 (2004)

[18] Michel, S., Triantafillou, P., Weikum, G.: KLEE: A framework for distributed top-k query algorithms. In: Proceedings of VLDB (2005)

[19] Vlachou, A., Doulkeridis, C., Nørvåg, K., Vazirgiannis, M.: On efficient top-k query processing in highly distributed environments. In: Proceedings of SIGMOD (2008)

[20] Zhao, K., Tao, Y., Zhou, S.: Efficient top-k processing in large-scaled distributed environments. Data and Knowledge Engineering 63(2), 315–335 (2007)

Paper G

The DASCOSA-DB Grid Database System

Jon Olav Hauglid, Norvald H. Ryeng and Kjetil Nørvåg.
In *Grid and Cloud Database Management*, Giovanni Aloisio and Sandro Fiore (Ed.), Springer-Verlag, 2011.

Abstract

Computational science applications performing distributed computations using grid networks are now emerging. These applications have new and demanding requirements for efficient query processing. In order to meet these requirements, we have developed the DASCOSA-DB distributed database system. In this chapter, a detailed overview of the architecture and implementation of DASCOSA-DB is given, as well as a description of novel features developed in order to better support typical data-intensive applications running on a grid system: fault-tolerant query processing, dynamic refragmentation, allocation and replication of data fragments, and distributed semantic caching.

G.1 Introduction

During the recent years, there has been a trend towards applications deployed on increasingly larger distributed systems with need for advanced data management. A prime example of such applications is computational science applications that uses advanced computing capabilities to understand and solve complex problems. Such applications frequently requires powerful computing resources, for example delivered through *grid computing services*.

While grid computing has gained maturity through the recent years, management of data in grid systems is less mature. Data storage and access is still mostly file oriented, and it is mostly left to users to manage files and their locations as needed. Although some support has emerged for metadata management, more advanced database features are not widely supported.

The goal of our research is a reliable *database grid*, with location-transparent storage, i.e., users/applications do not have to care about where data is stored and where queries are processed. The aim is sites cooperating on data storage and processing while retaining autonomy, i.e., a grid-wide database system. It is important to note how our context differs from more traditional approaches. The focus is on applications where large amounts of data is created and used on the same site, and where parts of the data, in particular summary data, are accessed by other grid participants.

An example of such applications is weather forecasting, where the national weather forecasting institutions have large amounts of locally collected data, do forecast, and make the resulting data available. They also store historical data. Both the summary data and historical data will be of interest to, and used by, other weather forecasting institutions and environmental researchers.

In this chapter we describe *DASCOSA-DB*, a distributed database system, which, in addition to providing location-transparent storage and querying, also includes novel features like efficient partial restart of queries and redistribution of query operators in the context of failure, dynamic refragmentation, allocation and replication of data fragments, and distributed semantic caching. A detailed overview of the architecture and the implementation of DASCOSA-DB is given, as well as a description of some of the features developed in order to better support typical

data-intensive applications running on a grid.

The rest of this chapter is organized as follows: In Sect. G.2 we give a short overview of other similar systems. In Sect. G.3 we present the system architecture of DASCOSA-DB. Sect. G.4 describes how data and metadata management is handled, and Sect. G.5 explains query processing, including semantic caching and partial restart of failed queries. Our distributed monitoring and management tool is described in Sect. G.6. An experimental evaluation of the system is provided in Sect. G.7. Finally, we summarize our work and describe future research directions in Sect. G.8.

G.2 Overview of Related Systems

Distributed databases and query processing is not a new field. For an introduction to distributed databases, we refer to [15]. A survey of distributed query processing is given in [12]. In this section, we will give an overview of systems that are similar to DASCOSA-DB. This includes both storage systems without query capabilities and query systems without storage capabilities, as well as complete database systems.

Much of the more recent work is based on peer-to-peer (P2P) networks, both unstructured and structured. Especially distributed hash tables (DHTs) have received much attention. A number of papers deal with focused issues such as query processing in DHT networks, including [2, 7].

OceanStore [13] is one of the storage systems without query capabilities. It provides an infrastructure for permanent storage and replication of objects, but no query system. Objects are accessed based only on their globally unique ID, and this ID has to be known in order to retrieve or update the object.

BigTable [5] is a large-scale distributed storage system with a model closer to relational databases. The storage model is similar to the relational model, but tuples are not stored or accessed as one unit. Instead, a row key and column key is used for both read and write operations. It does not provide more advanced query languages.

DASCOSA-DB does not provide its own storage infrastructure, but relies on an existing relational DBMS to store data. In that way, it is somewhat similar to the pure query engines that only provide a query processing service and no persistent storage.

Astrolabe [16] is one such system. Astrolabe is a distributed, hierarchical aggregation system designed for system monitoring. Astrolabe provides an interface that is similar to a database system, i.e., it provides SQL queries and standard database programming interfaces like ODBC and JDBC. To achieve scalability, updates are spread using a gossip protocol that guarantees eventual consistency. There is no guarantee that a client reads the most recent data, but if updates stop, all clients will eventually agree on the most recent value.

PIER [11] is a middleware query engine built on top of a DHT. PIER does not permanently store its data. Data sources publish their data in the DHT and update them regularly, and data that are not refreshed are removed. Typically, a PIER network will contain only object metadata (e.g., filenames, sizes, tags) and

a reference to the original data object. Clients will query the network to get the references to the objects of interest and retrieve the objects separately.

The difference between these query engines and DASCOSA-DB is that, although DASCOSA-DB has a middleware architecture like PIER, it provides persistent storage by using a local database on each site. It is not necessary to constantly republish data, as is the case with PIER.

Among the systems that provide a full DBMS, with both query processing and storage, are Hyperion [17], Orchestra [22] and Piazza [6]. All these systems allow each site to have its own schema, and use schema mediation techniques to allow cross-site querying. PeerDB [14] also falls into this category of systems with heterogeneous schemas, but the approach to schema mediation is different. Instead of relying on schema mediators, information retrieval techniques are used to find matching relations.

DASCOSA-DB does not use schema mediation. The systems mentioned above are meant to connect existing databases and provide a common query interface. Although DASCOSA-DB is a distributed database system with a high degree of site autonomy, it still behaves as one system, not many different systems with a common interface.

Other systems based on a common schema include APPA [1], Mariposa [21] and ObjectGlobe [4]. APPA provides a multilayered solution on top of a structured or super-peer P2P network, where the bottom layer is a simple key/value-store and the top level provides advanced services such as schema management, replication and query processing.

Mariposa is a distributed database system that uses economic models to solve optimization problems. Mariposa sites buy and sell fragments and bid for the execution of queries. The trading and bidding makes sure queries are answered efficiently and that data are moved closer to where they are needed.

ObjectGlobe is a distributed query processing infrastructure that allows users to combine data sources and query operators from different providers at different sites to perform queries. Sites can sell data, query operators, computing power or a combination of these. The client combines these resources to a full query pipeline.

AmbientDB [3] is probably the system that bears the closest resemblance to DASCOSA-DB. AmbientDB is a system designed to provide full relational database functionality for stand-alone operation in autonomous devices that may be mobile and disconnected for long periods of time, while enabling them to cooperate in an ad hoc way with (many) other AmbientDB devices. A DHT is used both as a means for connection peers in a resilient way as well as supporting indexing of data.

Like AmbientDB, DASCOSA-DB is also constructed as a combination of middleware and federated databases, connecting the local databases of each site. The key difference is that AmbientDB is a system for mobile devices, which have low computational power and may frequently be disconnected from the network, while DASCOSA-DB is designed for sites that have the computational power necessary to do query processing and more stable network connections. DASCOSA-DB is also based on a DHT, like AmbientDB and PIER. However, the DHT is only used as a metadata catalog. Query processing uses point-to-point links following the query

tree, more like Mariposa and ObjectGlobe. This is different from PIER, where the DHT is used extensively in query processing.

In terms of query capabilities, all sites of DASCOSA-DB are equal. There is no buying or selling of query operators or data. Data is fragmented, allocated and replicated according to the needs of the combined load of all sites, trying to keep the costs of network communication low. Query operators are shipped out to sites in order to minimize network costs by trying to perform most query operations on local data.

Many of the systems mentioned above support SQL-like querying and presents data similar to a normal relational database system. DASCOSA-DB is fully a relational database system that supports standard SQL.

A brief description of a DASCOSA-DB demonstration is given in [9].

G.3 System Architecture

In this section, the architecture of DASCOSA-DB is described. DASCOSA-DB consists of a number of autonomous sites connected to form a distributed database system. First is described how sites are connected, how data is distributed and how sites cooperate to execute queries and updates, and then the internal architecture of a single site is presented in more detail.

G.3.1 Distributed Architecture

DASCOSA-DB is designed as a middleware layer that binds together local DBMSs running on different sites to make a distributed DBMS providing location transparency. Fig. G.1 shows the distributed architecture of DASCOSA-DB, as a middleware connecting local databases and applications to provide access to a large, distributed database. All sites are autonomous. There is no single site that controls the distributed DBMS. In this way, the sites act together as a peer-to-peer network.

All sites connect to form a DHT. This DHT is used to store the distributed catalog, which contains information on all tables, table fragments, replicas and cache entries in the system. Currently, FreePastry[1] is used, but any other DHT may be used.

A new site wishing to join a running DASCOSA-DB system only needs to know the address of one connected site in order to join the DHT and thus be a part of the distributed database. When it has joined, it publishes information about its local metadata in the distributed catalog in order to make its local tables available to the rest of the system.

Sites communicate using messages. These messages can either be sent directly to a site if the address is known, or routed to the target site using the DHT routing mechanism. The latter method is used for catalog lookups and updates.

DASCOSA-DB supports the relational model and bases its storage on a local relational database management system. The current implementation uses JavaDB,[2]

[1] http://freepastry.org/

[2] http://www.oracle.com/technetwork/java/javadb/overview/

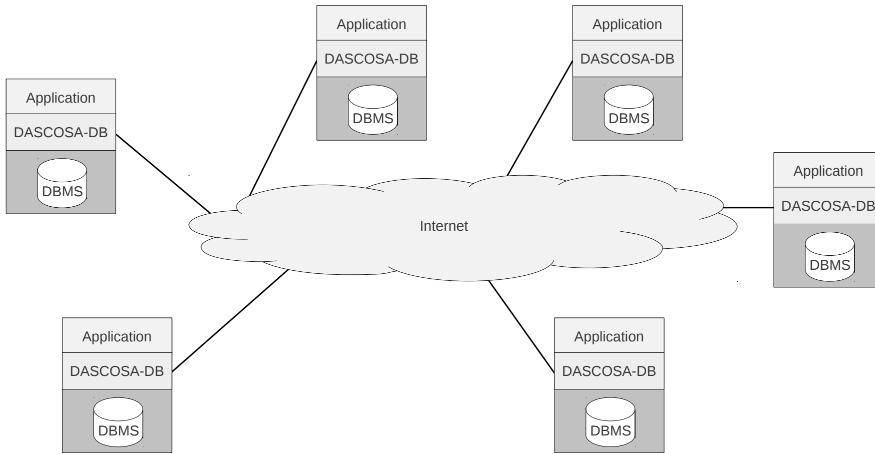


Figure G.1: Distributed architecture of DASCOSA-DB.

but any relational database management system may be used. The back-end database system can be chosen freely at each site.

The relational tables can be horizontally fragmented over a subset of the sites in the system. Each fragment can also be replicated. The distributed catalog maintains information about tables, fragments and their replicas. Creation and removal of fragments and replicas can be done using DASCOSA-DB's automated refragmentation method. Based on logging of read and write accesses, fragments can be split or joined, or replicas can be created and removed. This is done to reduce overall communication costs by making more data available locally where it is used and scale the number of replicas by the amount of writes. For example, a site doing heavy reads on a table fragment will get a local replica once this pattern is detected.

When executing queries, DASCOSA-DB utilizes query shipping. After query optimization, different query operators are allocated and distributed to sites in the system. This allows different operators to be executed by different sites in parallel. DASCOSA-DB also includes support for distributed semantic caching to speed up query execution. During updates, replicas are kept up to date using synchronous replication and transactions are handled using the two-phase commit protocol.

G.3.2 Site Architecture

The overall architecture of a DASCOSA-DB site is illustrated in Fig. G.2. As described above, sites communicate using direct messages or using the DHT. Together with modules handling broadcasting of messages to the network and request-response pairs of messages, these constitute the communication subsystem in DASCOSA-DB.

Local storage on a site consists of three parts. First, there is the relational data.

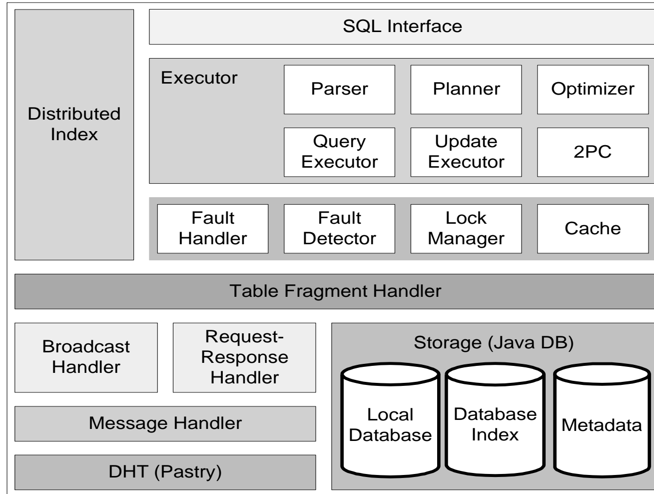


Figure G.2: High-level overview of the architecture of a DASCOSA-DB site.

Relational tables can have one or more fragments and each fragment has one or more replicas. Therefore, the unit of local storage is a table fragment replica. The second part of local storage is the indices for these replicas. Finally, each site stores a part of the distributed metadata catalog. Which part of the catalog a given site stores, is determined by the distributed hashing algorithm and the site's position in the DHT.

Which replicas a site stores locally can change at runtime. Based on an analysis of logged reads and writes, the local Table Fragment Handler can dynamically decide to change the fragmentation and allocation of replicas in one of four ways:

- Coalesce two fragments into one fragment. This means that all replicas of both fragments will have to be altered.
- Split a fragment into two fragments. As with coalesce, this will have global effect for all replicas of the fragment.
- Send a copy of a local replica to another site so that this site can get its own local replica to speed up local accesses.
- Delete a local replica. This will reduce the effort needed to keep all replicas of a fragment up to date and will therefore make sense in periods with many updates.

The Fault Detector and Fault Handler are used to implement partial restart of failed queries. If a site detects that another site designated to execute a subquery has failed, it can handle this fault transparently from the rest of the query execution. This is done by relocating the failed subquery to other sites. In many cases, this

can be done efficiently by not having the new sites restart the subquery completely, but rather continue where the failed site stopped.

Each site in the system can receive SQL queries and updates, for example using the provided user interface or using API calls. A received SQL statement is first parsed and transformed into relational algebra. If it is a query, a lookup in the distributed catalog is done to find location information about all involved tables. This information is then used by the Planner and Optimizer modules to generate a distributed query plan, including allocating the individual query operators to individual sites in the system. The operators are distributed to the involved site where the Query Execution module is responsible for the actual execution.

In order to facilitate easy interactive access to the system, as well as study configuration, distribution of data and query execution, DASCOSA-DB includes a monitoring tool that gives a live view of table fragments, replicas, catalog entries and cache entries. It also provides a live view of query execution, including network traffic and currently running query operators.

G.4 Distributed Data and Metadata Management

Tables in DASCOSA-DB may be horizontally fragmented based on the primary key, and DASCOSA-DB provides an adaptive fragmentation and replication system [10] that automatically moves data between sites as needed. In this section, the fragmentation process and then the replication of the fragments are described. Then it is described how metadata about fragments and replicas are retrieved from the local database when a site connects to the system and how it is published and subsequently retrieved from the global distributed catalog.

G.4.1 Fragmentation

A table may be stored in its entirety on one site, or it can be fragmented over a number of sites. An unfragmented table is treated as a table having a single fragment. Tables are fragmented horizontally based on the primary key. Each fragment of a table is given a fragment value domain (FVD) that defines which range of the primary key domain has been allocated to the fragment. The fragments are non-overlapping, and the FVDs of all fragments of a table cover the whole primary key domain.

The FVD of a fragment may cover a much larger range than the range of actual tuples in the fragment. E.g., a newly created table consists of one fragment with the whole primary key domain as its FVD, even though it does not store any tuples yet. As tuples are inserted, updated, read and deleted, a larger part of the FVD is actually used, and the table may split into more fragments

The traditional way of fragmenting and replicating tables in distributed database systems has been to use fixed value ranges or rules defined by database administrators. In DASCOSA-DB, fragments and replicas are created and migrated automatically by the system to accommodate the current query load. Based on access pattern monitoring, DASCOSA-DB will try to keep the number of accesses to remote sites

as low as possible. The FVDs and fragment placements are not fixed, so fragments can be split, coalesced and migrated automatically to adapt to changing workloads. Fig. G.3(a) shows a simple example of how two sites with different access patterns access the same table. Site S_2 has a few hotspots, while site S_1 accesses the whole table uniformly and infrequently. In this case, DASCOSA-DB will split (or merge if the table is already split) the table into 6 fragments, $F_1, F_2, \ldots, F_6$. F_1, F_3 and F_5 will be allocated to site S_2, while F_2, F_4 and F_6 will be allocated to site S_1.

In order to make informed decisions about useful fragmentation and replica changes, future accesses have to be predicted. As with most online algorithms, predicting the future is based on knowledge of the past. In our approach, this means detecting access patterns, i.e., which sites are accessing which parts of which fragment. This is done by recording accesses in order to discover access patterns. Recording of accesses is a continuous process. Old data is periodically discarded so that statistics only include recent accesses. In this way, the system can adapt to changes in access patterns.

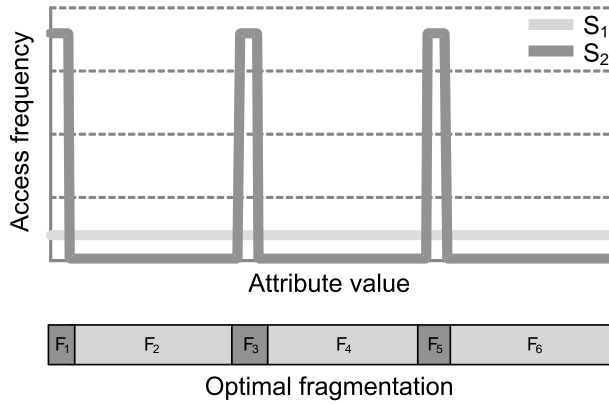
Given the available statistics, our algorithm examines accesses for each replica and evaluates possible refragmentations and reallocations based on recent history. The algorithm runs at given intervals, individually for each replica. Each site bases its decisions only on information available at that site, requiring no synchronization with other sites. With master-copy based replication, all writes are made to the master replica before read replicas are updated. Therefore, write statistics are available at all sites with a replica of a given fragment. On the other hand, reads are only logged at the site where the accessed replica is located. This means that read statistics are spread throughout the system. In order to detect if a specific site has a read pattern that indicates that it should be given a replica, it is required that each site reads from a specific replica so that each site's read pattern is not distributed among several replicas.

There is a great potential for cost savings by improving fragmentation. Fig. G.3(b) shows the reduction in number of tuples transferred over the network in DASCOSA-DB for two different workloads. In the general workload, all sites access tuples uniformly across a selected range of the whole table. 80% of the accesses are read accesses and 20% are write accesses. The reduction in tuple transfers is more than 40%. In the grid application workload, each site alternates between read phases and write phases, changing hotspots for each phase. The grid application workload has more clearly separated phases, and the savings are more than 50%. The results clearly show that the cost of splitting, migrating and replicating fragments pays off.

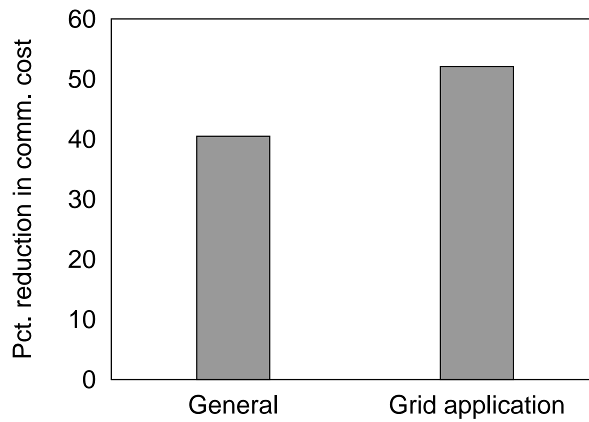
G.4.2 Replica Management

A table fragment is considered to be a logical entity. The physical entities stored in the local DBMSs are table fragment replicas. All fragments must therefore have at least one replica.

Replicas are kept up to date using synchronous replication. Every statement that changes the state of a fragment is sent to all sites with replicas. All replicas must be updated in order for a transaction to commit, and a two-phase commit



(a)



(b)

Figure G.3: (a) Access pattern and desired fragmentation. (b) Reduction in communication costs relative to static fragmentation.

protocol is used to ensure that all replicas agree on the decision to abort or commit the transaction.

Similar to the way fragments can be split or coalesced, replicas can be automatically created and deleted. Each site logs reads and updates to the locally stored replicas. A new replica is created at a given site if this site does a lot of reads. The idea is that the cost of transferring the replica to the site is less than having a constant stream of remote read requests. A local replica is deleted if there are few local accesses compared to the number of updates received. For both these mechanisms, the idea is to reduce the overall network traffic.

Not all replicas are treated equally. One replica is designated as the master replica. In order to ensure that automatic replica deletion does not delete all replicas, this replica is not eligible for deletion. The site containing the master replica has two special functions. First, it is the site where refragmentation decisions are made. This prevents two sites from independently and simultaneously deciding to, e.g., split the same fragment. Only the site with the master replica is able to do this. Second, the site with the master replica acts as a lock manager for the table fragment. This allows us to not have a centralized lock manager, which could become a bottleneck in a large system. When the system first boots, the catalog site storing the catalog entry for a table decides for each fragment of the table which replica becomes the master replica, and thus also which site becomes the master replica site. A new master replica site can be selected if the current master replica site crashes. It is also possible for the current master replica site to transfer this status in case of refragmentation.

G.4.3 Metadata Management

DASCOSA-DB uses a DHT to store and access the metadata catalog. The DHT provides a reliable and robust routing and lookup mechanism. Due to the DHT routing, catalog lookups are fault tolerant. The DHTs hashing function also distributes responsibility for metadata storage. All sites in the system participate in the DHT, and when a metadata object is published in the DHT, the DHT places it on one of the sites according to a hash of the object. Using a uniform hashing function, metadata objects are uniformly distributed among the catalog sites.

When a site joins the DHT, it scans its local database and inserts information on local objects into the DHT. Catalog objects will time out if they are not renewed, and sites periodically republish their information before the objects time out and are removed. This is done to ensure that erroneous information that may appear due to sites crashing after publishing their metadata is cleaned up regularly.

The catalog keeps track of tables and their schemas. For each table, it stores information about the primary key and the name and data type of all attributes. The catalog also keeps track of how tables are fragmented and replicated, i.e., how many fragments there are, the FVD of each fragment, and the number of replicas and their locations. Also, one replica of each fragment is designated the master replica, and the catalog stores this information.

The existence of caches of intermediate query results is also regularly published

to the catalog in the same manner as table, fragment and replica information. For each cached query result, the catalog stores a semantic descriptor, location and timestamp. Information about cache entries is not looked up directly, but rather discovered as a side effect of table lookups. The cache lookup is included in table lookup requests and replies. This mechanism is described in more detail in Sect. G.5.1.

G.5 Distributed Query Processing in DASCOSA-DB

DASCOSA-DB is a query shipping system where all sites store data and process queries. Queries may arrive from any site of the system, and the site that introduces a query to the system becomes the initiator site for that query. It is assumed that queries are written in some language that can be transformed into relational algebra operators, for example SQL.

G.5.1 Query Pipeline

A query enters the system at one site. This site, called the initiator site, becomes the coordinating site for this query. The initiator site transforms the query into an algebra tree. During query planning, the different algebra nodes are assigned to sites. This requires catalog lookups in order to transform logical table accesses into physical localization programs, e.g., a set of accesses to table fragment replicas. Sites can be assigned more than one algebra node so that one site can be assigned a whole subquery. As all sites have the capability to execute operators, sites storing table fragments used in the query are typically also assigned query operations on these fragments during planning. This tends to reduce network traffic as tuples can be processed locally. An example of an algebra tree with site assignment is shown in Fig. G.4(a). The initiator site plays the role of coordinator for this query and executes an initiator algebra node that is the endpoint of the query result.

DASCOSA-DB can cache the intermediate and final results of queries. Each site autonomously caches results of locally executed queries and subqueries and registers these in the distributed catalog so that the caches can be found by other sites. These catalog entries contain a semantic description of the cached query result, the address of the site that stores the cache entry, and a timestamp used to check cache entry validity.

As Fig. G.4(a) indicates, the complexity of a query increases with the height of the query tree. The query $T * U * V$ is more complex than $T * U$. If some of the intermediate results, like $T * U$, are cached, the more complex queries may be answered partly from these caches, saving both execution time and computational cost. More complex results in cache means larger savings when these caches are used. However, as the other arrow in Fig. G.4(a) shows, the reusability of a result is higher for the less complex queries.

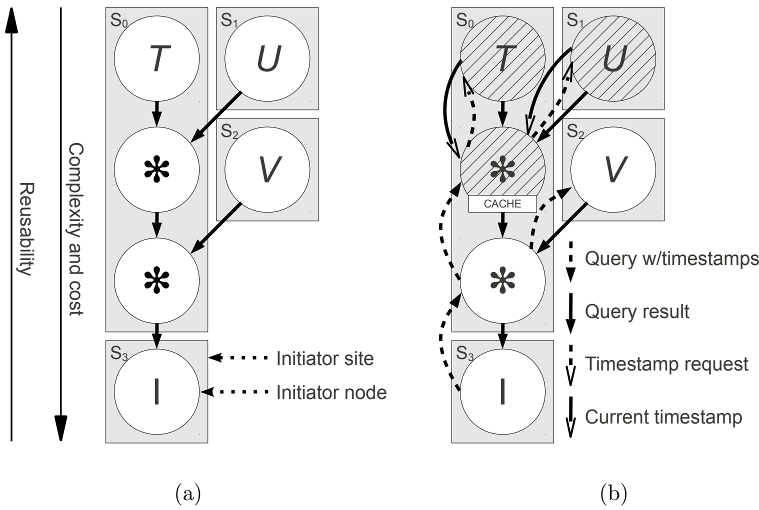


Figure G.4: (a) Example query plan. (b) Query dissemination with a cache hit.

When a table is looked up in the catalog, the initiator site piggybacks a representation of the query to the lookup message. The catalog site that handles the lookup request sees this query representation and responds by piggybacking onto the response a list of suitable cache entries that might speed up query processing. Information about a cache entry is stored at the same site as one of the tables involved in the query that produced it. This means that after looking up all tables, the initiator site has been told about all caches involving the combination of these tables. During localization, the initiator site looks at these cache entries. If a relevant cache entry is found, the initiator site can rewrite the query to use the cache entry. This is done by including the query that produced the cached result as a subquery of current query and assigning the subquery to the site where it is cached.

After planning and possibly rewriting the query to use cached intermediate results, query dissemination begins by transmitting the algebra tree stepwise from the initiator site to the different sites involved. The root algebra node always stays at the initiator site. For each child of the root node, the initiator site sends out the subtree rooted at that child node to the child's assigned site. These sites, upon receiving query subtrees where the roots are assigned to them, loop through the children of the roots and ship them off to the sites to which they are allocated. This continues until all nodes have reached their destination. The result of this stepwise transmission is that each site knows the complete subquery for which it is the root.

However, if a site receives a subtree for a query it has in its cache, and if that cache entry is still valid, further dissemination of that subtree stops. Instead, the site prepares a special algebra node to produce the result from cache. To the sites higher up in the hierarchy, there is no way to tell if the result is served from cache or produced from scratch. This transparency allows the sites to make cache decisions without relying on central coordination. Fig. G.4(b) shows query $T * U * V$ with

a cache hit on subquery $T * U$. Site S_0 checks the timestamp of the cache entry against the timestamps of T and U to see if the cache is up to date. If it is, $T * U$ is delivered from cache, and the only query operator actually executed is the join of $T * U$ and V at site S_0. Site S_1 is never involved in the query processing, except when replying to the request for the timestamp of U.

Results of query operators are transferred between sites in tuple packets. The system supports stream-based processing of tuples, for example joins performed by pipelined hash-join [23]. This means that an algebra node usually can start producing tuples before all the tuples are available from its operand nodes. This makes it possible for nodes downstream to start processing as soon as possible and therefore lets more nodes execute in parallel. This requires each site to be able to accept and buffer yet unprocessed packets, but it allows data transfers to be made without explicit requests, thereby improving response time. In case of limited buffer availability, flow control is used to temporarily halt packet transmissions.

The result of any algebra operator is a candidate for caching at the site where it is produced. Sites are allowed to use any cache replacement algorithm they want. A cache entry is usable as soon as it is created, but in order to enable the query planners to plan on using cached results the cache entries must be registered in the distributed catalog. A site that has cached a result reports its existence to the same site that handles lookup request for one of the tables used to produce the result. E.g., if the cache entry is the result of $T * U$, the catalog stores the information about this entry at either the site that stores the catalog entry for T or the catalog entry for U. Any site that later looks up both T and U in order to perform a join is guaranteed to find this entry.

G.5.2 Standard Query Operators

DASCOSA-DB supports the typical query operators. At the lowest level, the scan operator accesses the local DBMS and delivers tuples of a table fragment. In order to speed up execution, special scan nodes exist that push selection and projection down into the local DBMS.

Selection and projection operators also exist to be inserted into the query tree when the operations cannot be pushed down into the local DBMSs. The selection operators also support set operators, i.e., IN and EXISTS, to compare against the result of subqueries.

The join operators include natural join, equijoin and outer join. These are implemented as pipelined hash joins. An operator also exists to produce the Cartesian product. Other operators include sorting, limiting, aggregation (including grouping), duplicate removal (UNIQUE) and a skyline operator.

All operators, except the scan operators, have flow controlled input and output streams with a general interface. This makes it possible to connect them in any meaningful way to represent a query. This generalized interface also makes it easy to ship queries around, since the input and output streams are network transparent.

For most normal cases in-memory operators suffice, but for large operand sizes there are also variants of these operators that will use disk to avoid excessive memory

consumption.

G.5.3 Fault-Tolerant Distributed Query Processing

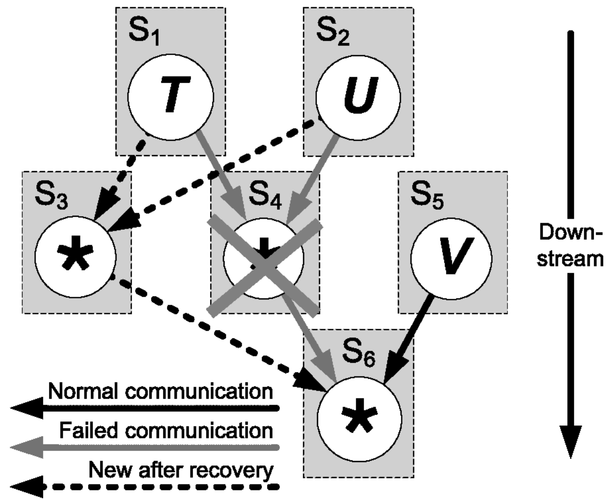
The more sites that are involved in a query, the higher the probability of a site failing during query processing. Long queries and high churn rates in the system also increases the probability of site failures. The traditional way of handling failures focuses on update transactions, and the typical failure recovery is to do a complete restart of the failed transaction. Query failures have largely been overlooked. Complete query restart is an appropriate technique for small and medium-sized queries, however it can be expensive for very large queries, and in some application areas there can also be deadlines on results so that complete restarts should be avoided. In some cases, various checkpoint-restart techniques have been employed to avoid complete restarts of operations, but these techniques have been geared towards update/load operations, and in many cases implies that a query will be delayed until the failed site is back online.

As an alternative to local checkpointing and complete restart, DASCOSA-DB supports partial restart of queries [8]. With partial restart, unfinished subqueries from failed sites can be resumed on new sites after failures. These restarted subqueries may also utilize partial results already produced before the failure — both results generated at non-failing sites and results from failing sites that have already been communicated to non-failing sites. The technique integrated in DASCOSA-DB can be compared to previous approaches like [20]. DASCOSA-DB's fault tolerant query processing 1) reduces query execution time compared to complete restart, 2) incurs minimal extra network traffic during recovery from query failure, 3) employs decentralized failure detection, 4) supports non-blocking operators, 5) handles recovery from multi-site failures, and 6) avoids duplicate tuples by deterministic delivery of tuples from base relations and operators. The query restart techniques can also be used to provide distributed suspend and restart of queries.

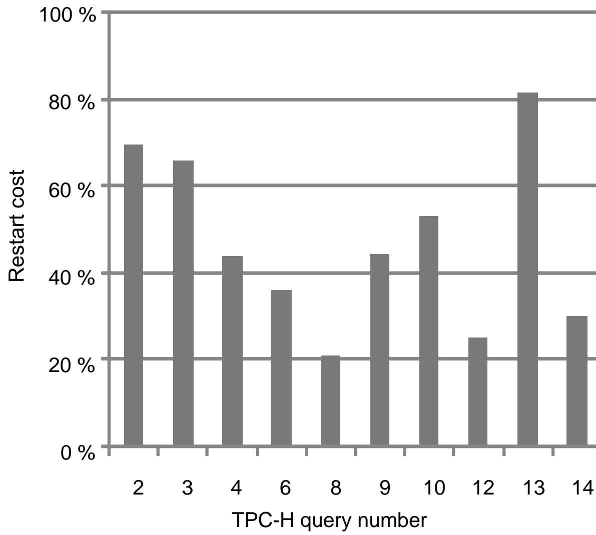
Fig. G.5(a) shows a system executing the query $T * U * V$. Originally, only sites S_1, S_2, S_4, S_5 and S_6 are involved, but sometime during query processing S_4 fails. This is detected by site S_6, which is the recipient of the result of the failed algebra node. Site S_6 chooses S_3 to replace S_4, and reissues the query $T * U$ to this site. Site S_3 follows the normal query dissemination strategy and forwards the scan operators to sites S_1 and S_2. The particular challenges that have been solved in our approach relate to failure detection, selection of replacement site, and restart of the various relational algebra operators.

Failures during query processing are detected by using timeouts. There is no central failure detector. Instead, a site monitors all sites that produce the operands for query operators executing at that site. If a site failure is detected, a new site is selected for each of the failed operators. The impact of a failure is therefore localized — it only affects the sites receiving the results of the failed query operators. Other queries and subqueries executing at other sites continue as normal.

The replacement site selected to execute a failed query operator tries to pick off where the operator first failed. How this is done, depends on the operator. Two



(a)



(b)

Figure G.5: (a) Example of query failure and restart. (b) Relative cost of restarted TPC-H queries.

classes of operators can be identified: *stateless* and *stateful*. Stateless operators process tuples independently. Examples include projection and selection. For these operators, the number of operand tuples an operator has used to produce a given number of result tuples is stored. This number is transmitted with each packet of tuples sent in the network. Using this number, a replacement site knows where to start when resuming a failed operator. For example, assume that a failed site S_f was executing a selection. This selection was done on tuples received from another site S_o. The target site S_t for the selection, has received 500 result tuples when S_f fails. Assume that 800 tuples from S_o had been processed to produce those 500 result tuples. This fact will be known by S_t and transmitted to the replacement site S_r. S_r will then know that it should request S_o to resume sending tuples, skipping the first 800.

For stateful operators, on the other hand, each result tuple can be dependent on more than one operand tuple. Such operators include join and aggregation. When such operators are restarted, they must request operands to be replayed in full. However, they can still use the number of received operands before the failure to prevent sending duplicates. E.g., a join must get its two operands completely, but it can skip sending the first result tuples up to and including the number of tuples received from the failed site.

For this partial restart technique to work correctly, tuples must be produced by an operator in a deterministic order. Note that this does not mean that is has to be a sorted order. For scan operators, it is required that tuples are retrieved from the local DBMSs in a deterministic order. Further, it is required that other operators are deterministic so that they produce tuples in a deterministic order given the same ordering of operand tuples. Thus, this requirement reduces to having operators consuming tuples in a deterministic order. This is achieved by having operators consume packets of operand tuples in a round-robin order sorted on the ID of the source site of an operand tuple packet.

The results in Fig. G.5(b) show the cost of a restart for a representative collection of ten TPC-H queries. The average restart cost is 50%. The two queries with the least gain (query 2 and 13) were also the two shortest queries. There is a constant overhead in detecting site failure and restarting queries. For the longer queries, this constant overhead is relatively small, so these queries have a lower restart cost.

G.6 Distributed Monitoring and System Management

DASCOSA-DB includes an integrated distributed monitoring and management tool. Fig. G.6 shows the user interface which allows the user to issue SQL statements and monitor the state of the system in real time. It has proven very useful for the different research projects employing or extending DASCOSA-DB.

DASCOSA-DB supports running more than one site on the same physical computer. All these sites will still communicate as if distributed and have separate local DBMSs. Running more than one site locally allows the user to easily examine the

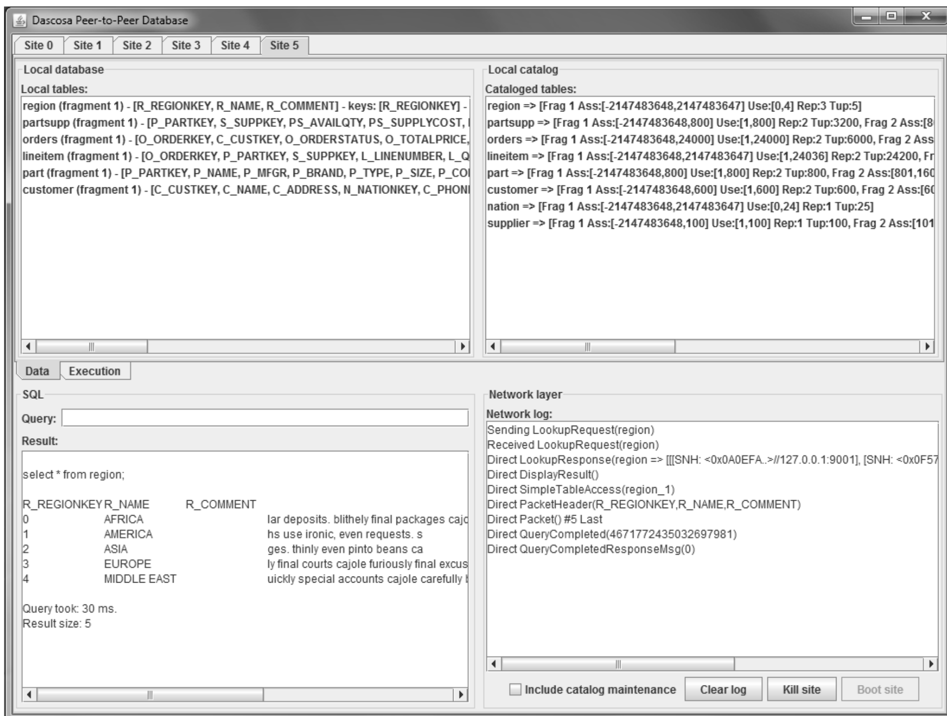


Figure G.6: Screenshot from the DASCOSA-DB system monitoring tool.

execution of distributed queries as the monitoring tool can observe all these sites.

The available views show which table fragments are stored at each site and the schema for each of these. The catalog view for a site shows catalog entries stored at that site. Tables are listed with the number of fragments and replicas, and each fragment entry shows the FVD, the actual used ranged and the number of tuples in the fragment. The catalog view also shows cached query results.

Network traffic monitoring is made easy by using the network log, which will list all messages received and sent by a selected site. This allows the user to, e.g., easily track the distributed execution of a query. Both query processing messages, catalog messages and other maintenance messages can be inspected.

The monitoring tool also allows the user to inspect running queries and follow the execution of algebra nodes as flow control changes the state of algebra nodes between processing and paused states. A complete view of all running queries and algebra nodes is provided.

Cache inspection is also provided. DASCOSA-DB has two caches: a restart cache that is used to provide fault tolerant query processing, and a semantic cache of intermediate query results. Each of these may be inspected through the management interface.

Finally, the management interface allows the user to simulate network failures and site crashes by toggling on or off message delivery to each site. When a site is disconnected, the rest of the system will notice its disappearance and adjust to the new situation. Queries involving the failed site will restart, and new master replicas will be appointed.

G.7 Experimental Evaluation

The individual features of DASCOSA-DB have been evaluated experimentally in earlier papers [8, 10, 18]. In this section, it is showed how the system, as a whole, scales. Evaluation of additional DASCOSA-DB features not described in this chapter can be found in [19].

G.7.1 Experimental Setup

The system consists of 10 interconnected sites running DASCOSA-DB. A TPC-H dataset is horizontally fragmented into five fragments. Each site stores one fragment, meaning that there are two replicas of each fragment. A set of 1000 random TPC-H queries with random values for substitution parameters is used. An 80/20 distribution is used both for query and parameter selection.

The number of sites that issue queries, and thereby the number of coordinator sites, is varied between 1, 5 and 10 to show how system performance increases with increased parallelism. Each querying site executes its queries in series, waiting for one to complete before issuing the next. The system is tested both with and without semantic caching enabled.

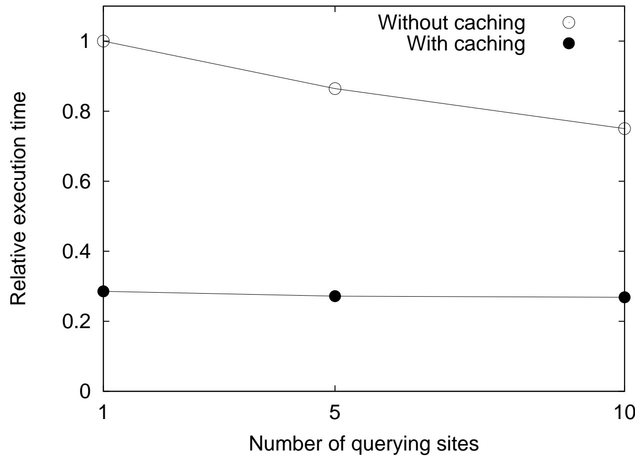


Figure G.7: Execution time relative to baseline.

G.7.2 Results

The execution time of each experiment relative to a baseline is measured, where all queries were issued in sequence from a single site, without caching any query results. The results shown in Fig. G.7 show that by increasing parallelism so that all sites issue queries, execution times are reduced by 25%. Since DASCOSA-DB allows queries to be issued from any site, the risk of the coordinator site becoming a bottleneck is reduced, and higher throughput can be achieved.

Further, semantic caching reduces the run time with up to 73%. This considerable improvement is possible because parts of the algebra tree for a query is similar to some parts of other queries. These parts are reused to provide a quicker response to the query, freeing up resources that otherwise would be used to process each query from scratch.

The execution time does not decrease as much with increasing number of querying sites as was the case without caching. The reason for this is that there is not much more time to save after the reduction in execution time caused by semantic caching. Also, caching is a means to improve execution time of a series of queries, not parallel queries. The result has to be cached before it is used. Still, our semantic caching method makes it possible to reduce execution time of multiple parallel querying sites since cache entries are shared with all other sites.

G.8 Summary and Future Challenges

The central point of the grid is to present the user with readily available computational power without the need to know where this power comes from. This should also be the central point for data storage used by the grid, and our DASCOSA-DB is designed with this in mind.

We have presented a middleware system that transparently provides access to data distributed throughout the grid. Based on the relational model, our query shipping database system efficiently queries data in situ, while constantly adapting to the shifting workload by moving table fragment replicas closer to where they are used and by replicating data that has to be read by many sites. Semantic caching reduces the need to compute everything from scratch and allows new queries to take advantage of the intermediate results of queries that have already finished, even if they came from different sites. In case of failures during query processing, DASCOSA-DB will restart only the failed subquery. DASCOSA-DB also provides a distributed monitoring and management system.

Although we now have a working distributed database system, there is no lack of remaining challenges. More advanced optimization in the presence of cached data is needed. We will also study rank-aware operators which are important for many of the intended application areas.

Acknowledgments

The development of the DASCOSA-DB has been supported by grant #176894/V30 from the Norwegian Research Council.

Bibliography

[1] Akbarinia, R., Martins, V., Pacitti, E., Valduriez, P.: Design and Implementation of Atlas P2P Architecture. In: Global Data Management, 1st edn. IOS Press (2006)

[2] Bauer, D., Hurley, P., Pletka, R., Waldvogel, M.: Bringing efficient advanced queries to distributed hash tables. In: Proceedings of LCN (2004)

[3] Boncz, P., Treijtel, C.: AmbientDB: relational query processing in a P2P network. In: Proceedings of DBISP2P (2003)

[4] Braumandl, R., Keidl, M., Kemper, A., Kossmann, D., Kreutz, A., Seltzsam, S., Stocker, K.: ObjectGlobe: ubiquitous query processing on the Internet. VLDB Journal **10**(1), 48–71 (2001)

[5] Chang et al., F.: Bigtable: A distributed storage system for structured data. In: Proceedings of OSDI (2006)

[6] Halevy, A.Y., Ives, Z.G., Madhavan, J., Mork, P., Suciu, D., Tatarinov, I.: The Piazza peer data management system. IEEE Transactions on Knowledge and Data Engineering **16**(7), 787–798 (2004)

[7] Harren, M., Hellerstein, J.M., Huebsch, R., Loo, B.T., Shenker, S., Stoica, I.: Complex queries in DHT-based peer-to-peer networks. In: Proceedings of IPTPS (2002)

[8] Hauglid, J.O., Nørvåg, K.: PROQID: Partial restarts of queries in distributed databases. In: Proceedings of CIKM (2008)

[9] Hauglid, J.O., Nørvåg, K., Ryeng, N.H.: Efficient and robust database support for data-intensive applications in dynamic environments. In: Proceedings of ICDE (2009)

[10] Hauglid, J.O., Ryeng, N.H., Nørvåg, K.: DYFRAM: dynamic fragmentation and replica management in distributed databasesystems. Distributed and Parallel Databases **28**(2–3), 157–185 (2010)

[11] Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the internet with PIER. In: Proceedings of VLDB (2003)

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

[13] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., Zhao, B.: OceanStore: An architecture for global-scale persistent storage. In: Proceedings of ASPLOS (2000)

[14] Ng, W.S., Ooi, B.C., Tan, K.L., Zhou, A.: PeerDB: A P2P-based system for distributed data sharing. In: Proceedings of ICDE (2003)

[15] Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice-Hall (1991)

[16] van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. **21**(2), 164–206 (2003)

[17] Rodríguez-Gianolli et al., P.: Data sharing in the Hyperion peer database system. In: Proceedings of VLDB'2005 (2005)

[18] Ryeng, N.H., Hauglid, J.O., Nørvåg, K.: Site-autonomous distributed semantic caching. In: Proceedings of SAC (2011)

[19] Ryeng, N.H., Vlachou, A., Doulkeridis, C., Nørvåg, K.: Efficient distributed top-k query processing with caching. In: Proceedings of DASFAA (2011)

[20] Smith, J., Watson, P.: Fault-tolerance in distributed query processing. In: Proceedings of IDEAS (2005)

[21] Stonebraker et al., M.: Mariposa: A wide-area distributed database system. VLDB J. **5**(1), 48–63 (1996)

[22] Taylor, N.E., Ives, Z.G.: Reliable storage and querying for collaborative data sharing systems. In: Proceedings of ICDE (2010)

[23] Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases **1**(1), 103–128 (1993)