NTNU

Norwegian University of
Science and Technology

# Semantic Cache Investment

Adaption of Cache Investment for DASCOSA

**Konrad Giæver Beiske**
**Jan Bjørndalen**

Master of Science in Computer Science

| | |
|---|---|
| Submission date: | June 2009 |
| Supervisor: | Kjetil Nørvåg, IDI |
| Co-supervisor: | Jon Olav Hauglid, IDI |

Norwegian University of Science and Technology
Department of Computer and Information Science

# Problem Description

This master's thesis will look into how information about semantic cache can be used in optimization of queries for distributed databases. Information such as location and cost shall be used in this process. Implement a solution that evaluates the profit of suboptimal choices in order to get more valuable contents in cache. Adapt and implement the caching strategy called cache investment to work with the peer-to-peer database management system DASCOSA.

Assignment given: 19. January 2009
Supervisor: Kjetil Nørvåg, IDI

## Abstract

Semantic cache and distribution introduce new obstacles to how we use cache in query processing in databases. We have adapted a caching strategy called cache investment to work in a peer-to-peer database with semantic cache. Cache investment is a technique that influences the query optimizer without changing it. It suggests cache candidates based on knowledge about queries executed in the past. These queries are not only limited to the local site, but also detects locality in queries by looking at queries processed on remote sites. Our implementation of Semantic cache investment for distributed databases shows a great performance improvement, especially when multiple queries are active at the same time.

To utilize cache investment we have looked into how a distributed query optimizer can be extended to use cache content in planning. This allows the query optimizer to detect and include beneficial cache content on remote sites that it otherwise would have ignored. Our implementation of a cache-aware optimizer shows an improvement in performance, but its most important task is to evaluate cache candidates provided through cache investment.

# Preface

This paper is the result of a master thesis in computer science during the spring of 2009. The title of the project is "Semantic Cache Investment", and it was given by the Database Systems Group at the Department of Computer and Information Science. This master thesis was the continuation of our work in "DASCOSA Query Optimizer" during the autumn of 2008. Participants on this master thesis was the two final year master students, Jan Bjørndalen and Konrad G. Beiske.

We would like to thank our supervisor, Jon Olav Hauglid, for providing us with solid guidance throughout the whole period. He was of great help during research and implementation, and gave good feedback during the creation of this thesis. We would also like to thank the rest of the DASCOSA team, Kjetil Nørvåg and Norvald Ryeng for the aid they have given us with their expertise during our work.

June 15, 2009

Jan Bjørndalen          Konrad G. Beiske

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Databases are used all over the world to help people handle their data. Such databases are usually run centralized on a single computer. Accessing data in a database is just a matter of formulating a question in a query language, such as SQL, which enables the database to look for the data on its own terms. The effort required to process a query closely depends on how the database decides to run the query. This is where the the query optimizer comes into play. The query optimizer is tasked with finding the most efficient ordering of operations in a query, a query execution plan. Several plans are generated consisting of operations interpreted from the query language. The goal is to keep the cost of executing the plan low, and the best plan is the plan evaluated to the lowest cost among all possible plans for a given query. Good query optimization is important, because without it database management could not work in an efficient manner. If results are not produced according to time and demand, they are useless. Query optimization for single-computer databases are a mature topic where much research has been done[1][2][3].

Distributed databases is a research subject with a newfound interest. New generations of computer network technology have made this concept possible. Distributed databases introduces the location of data as a new factor. The need to use data at more than one location, raises new issues when dealing with resources, scalability and availability requirements. The important question is, how these new challenges can be addressed and bring distributed databases to the same mature state as the centralized databases. In a distributed database system the computer network is utilized to connect multiple databases[4]. There are several motivations for a database to be distributed over more than one computer. One computer neither has enough resources to handle the load or the preferred safety in case of system crash. Multiple contributors of data will ultimately lead to larger data collections than one machine can handle alone. How network communication is handled should also be considered to achieve quick response and good throughput. More machines holding the data results in higher redundancy, because in case of data loss the data could be recovered from another copy in the system. This would lead to a greater chance of surviving a failure. Keeping data safe and uncorrupted is a very important issue of data management.

The primary drawback of distribution is added complexity and the complications that follow[1]. Examples of such are maintaining ACID-properties, scalability, heterogeneous systems and utilizing all resources with as little shared state as possible. Distribution and scalability is an issue that will directly challenge many old assumptions in query optimization. These complications will need to be addressed before distributed database systems can reach full efficiency. In order to maintain the autonomy and independent access that have made relational databases popular in the first place[5], updates will have to be done to the now distributed database management system (DBMS).

Data transfer is in many distributed database systems the dominating contributor to both the total workload and the response time of individual queries[1]. As there usually is more than one way to execute a query, the system should choose the less costly one. One basic approach is to reduce the temporary result as early as possible, which in turn often reduces the amount of data transfered. However the cost of suboptimal choices in one operation might give greater payoff later on. Hence, there is a need to look at the complete problem space, which is often too large to handle without compromising between precision and cost. Having a good query execution plan will have a great impact on execution time. Distributed query optimization is even more important than centralized query optimization, because the extra options provided allow for more execution plans. The extra factors from distribution causes the variation between the best and the worst plan to be even larger.

"Hitting the memory wall"[6] describes a consequence of Moore's law that makes computing power increase faster than memory bandwidth. Whether the authors of [6] were correct in their predictions of future processing power ultimately becoming limited by the memory, is outside the scope of this thesis. However, there remains little doubt in the scientific community that better caching alleviates the problem[7]. In the research field of DBMSs there has been a great focus on minimizing data access, often at the cost of using more CPU resources. We expect caching between sites to be necessary in order for distributed database systems to reach their full potential. The question on what to cache in a distributed DBMS is however not as simple as for its centralized case. When other sites make decisions on what to cache the value of local caching candidates changes. Typically there is no need for two copies on the same network cluster.

Cache in databases can be divided into data cache, which is the traditional way of caching, and semantic cache. Semantic cache is a new caching technique that utilizes a semantic description to describe its content. This frees the cache from the overhead of index maintenance and gives it more flexibility[8]. The focus of this thesis will be on semantic cache and its use in databases.

In order to be able to cache data, the data must be produced so that the cache module can create a cache entry for it. Prediction of future queries based on previous queries is useful in identifying this data. Even though an entry is useful to a query, it does not imply that the entry will be created by that query. The example used by [9] is when a client has to choose between performing a selection on the server or locally. Doing the selection at the server implies less data transfer, because in order to do it locally it has to ship the entire table in stead of just the result. The optimizer would then choose to do selection at the server. If it had done the selection locally, the table would be in the local cache and future queries doing other selections on the same table would not require any data transfer at all. Cache investment works by using the history to decide what should have been cached and then tells the optimizer that it is cached[9]. This allows for creating cache entries that only pay off for more than one query, by influencing the query optimizer to produce certain plans that are identified to contain valuable cache content. The adaption and implementation of cache investment for DASCOSA will be the main focus of this master thesis.

## 1.1 Problem Description

The focus in this master thesis is how we can improve the use of cache for query processing in DASCOSA. DASCOSA is a distributed DBMS developed to facilitate research on the subject of distributed databases. The architecture of DASCOSA is based on the P2P paradigm. DASCOSA is using a caching method called semantic cache, which augments the cached data with a semantic description.

The content of semantic cache is the intermediate results of previous queries, which changes the way the cache is applied during query processing. Instead of acting as a substitute to data fetching, semantic cache is more often used as a substitute to more complex sub-queries. Because of this the usage of semantic caching must be addressed differently.

The P2P nature of DASCOSA complicates the process of using cache even further because DASCOSA, with its autonomous sites, does not have full knowledge of the system state. This limits the number of metrics available for the query optimizer to work with.

Currently the query optimizer does not use cache during planning. Cache is only used during execution if an exact match between a sub-query and cache data can be found. In such a case the database runs the risk of not using the cached data as often as it could. The process of doing better cache utilization has two sides.

The query optimizer must be adapted so that it becomes aware of cache that is available on its own and remote sites. We expect to show, that actively including cached data during planning will increase performance gain and improve the utilization of said cache.

The content being cached should be subject to improvement. We will adapt a caching technique called cache investment that suggests good candidates based on previously logged queries. The query optimizer will play an active role in deciding if the suggestions provided through cache investment will be profitable for the system to cache. We will rely on the query optimizer's knowledge about network topology. This knowledge will aid in determining the best location to establish cache and where would be the best location to retrieve cached data from during planning.

## 1.2 Our Contribution

In this project we describe a solution based on cache investment, adapted to work with semantic cache in a distributed database. The solution will be made as an extension of the query optimizer we designed for the DASCOSA DBMS[10].

Specifically, our solution provides.

- Cache-aware optimizer that is able to include cache content on remote sites during planning.
- Semantic cache investment for a peer-to-peer environment. Queries are logged to the distributed index and are thereafter subject to a method that identifies good cache candidates.
- A new structure for representing queries that makes two query trees comparable even if their subtrees have differences. We call this a Result Set Identifier for its ability to represent intermediate query results.
- Method for suggesting the best site for a cache candidate, based on the use of network coordinates. Network coordinates was introduced in our implementation of the DASCOSA query optimizer.
- Performance evaluations for our cache-aware optimizer and cache investment. Both solutions has been implemented in the DASCOSA DBMS.

## 1.3 Approach

We will approach this problem by creating a modular solution that will enhance the performance of caching in query processing. As a base for this work we will use the DASCOSA database to implement and evaluate our solutions. DASCOSA already has support for semantic caching, which we will refer to as implicit caching. Our first extension will be to add to the query optimizer the capability to plan with cache. This we will refer to as explicit caching. The second extension will be to add more functionality to the query optimizer with new operators such as selection and project. These operators will allow the query optimizer to utilize more specialized cache entries. The third extension will be an implementation of cache investment for semantic caching.

These three extensions will be referred to as the three phases in this project. Phase 1 will be known as 'Making the Optimizer Cache-Aware', phase 2 as 'Extending Operator Support', and phase 3 as 'Cache Investment'. This arrangement of phases will be applicable both during Design and Implementation and Evaluation.

Phase 2 is a direct consequence of scoping done during our implementation of the DASCOSA query optimizer described in [10]. The DASCOSA query optimizer is an exhaustive cost-based query optimizer which merges the ideas of R*[11] with the concept of network coordinates for sites[12]. Join operations are considered the most costly elements in query execution and was given priority due to our set time frame. The optimizer currently considers join ordering and site selection for joins and table scans[1]. This project will implement more operators for query planning. The scope for phase 3 on the other hand will be limited to only create hints for natural joins of tables.

Finally, we will evaluate our implementation by running queries from TPC-H[13] with different network topologies. Based on this we will consider if cache investment is suitable for a semantic cache in a distributed DBMS, and in which settings its performance differs. To do this we will measure the change in query response time for cases with different caching strategies applied.

## 1.4 Outline

Here is a description of the chapters in this document.

**Preliminary Study** This chapter explains the theory that will be helpful in understanding the problem and our solution. Among the subjects described are distributed databases, queries and query optimization, caching, using cache and cache investment. We will also give a introduction to the DASCOSA database, which we will use as a base for implementing and evaluating our solution.

**Design and Implementation** This chapter will describe our solution in detail. We divide the work into three phases. The phases are not standalone solutions, and build on work done in earlier phases. The work in each phase will be split into a design section where we describe our plans for the phase, and an implementation section where we describe how we achieved our goal. Special attention will be given to the pros and cons of the data structure invented to solve the problem.

**Evaluation** This chapter will describe the evaluation of our solution. An understanding of our testing environment, test cases and evaluation criteria

[1]DASCOSA supports table replication.

will be given. Each test case and its results will be explained in detail.

**Conclusion** This last chapter will conclude our work with our findings. We will give a summary of the merit of semantic cache investment in DASCOSA and how it performed compared to other caching strategies. We will explain which goals that has been achieved in this project, and what has been reserved to be solved in the future.

# Chapter 2

# Preliminary Study

In this chapter we will detail the relevant existing theory that our research is based upon. To start with we will introduce the concept of distributed databases, and how these differ from the traditional centralized databases. We will describe queries and the function of queries in a database.

Then we will give a brief introduction to caching before continuing with query optimization, before we will come back to the topic of cache optimization. Query optimization has been described in greater detail during our work on the DASCOSA query optimizer[10], but we will give an introduction in this chapter. Query optimization with cache is the main focus of this project. We will describe how the query optimizer and cache can work together to create a synergy effect on performance in a distributed database setting.

Then we will introduce an advanced caching strategy, cache investment. Cache Investment is an opportunistic caching strategy that actively tries to influence the query optimizer to make sub-optimal plans that will generate better cache entries.

In the end of this chapter we will describe the architecture of the DASCOSA database which we will use in Chapter 3, Design and Implementation, to develop our solution.

Among work related to ours we count the original cache investment design for client/server architecture[9], the ADMS query optimizer which integrate query result caching and matching[14], query optimization with materialized views[15] and answering queries by semantic views[16].

## 2.1 Distributed Databases

Database management systems (DBMS) have been popular for some decades now[17]. There are several reasons for this. An autonomous DBMS with a declarative query language, like the Structured Query Language (SQL), provides data independence from the systems accessing the data. Having several different systems access the same data concurrently with high throughput is a tough task while maintaining ACID-properties of every transaction. DBMSs have matured the techniques to address all these issues. To use a standard of-the-shelf DBMS is in many cases an obvious choice. Such systems have traditionally been centralized DBMSs.

For many organizations this task cannot be done by one machine. For instance, the total load might simply be too large for the biggest commercially available computer to this date. The systems accessing the data might be geographically distributed and the connecting network inadequate for a centralized solution. Availability requirements may also mandate a distributed solution as one computer alone represents a single point of failure. Independent data access, making all data available irregardless of location, is a commonly desired property of a distributed DBMS and an important one in making a cluster of sites participating in a DBMS appear as one site to the user.

## 2.2 Queries

A query is a request to the database for information. Database queries are well formed, based on the query language used, such as SQL. Once in the DBMS the query is translated into relational algebra[1]. Relational algebra consists of a series of operators. There is one operator type for all tasks needed to transform the data. Together these operators produce the result for the query. Relational algebra operators can be represented as a tree. The leaf nodes are data relations. Each inner node represent some job that must be done on the data to produce the result, such as a constraint or join. Typically two or more data relations are combined through a join to produce a new data set which are handed over to the next relational algebra operator on the next level in the tree.

```
SELECT *
```

```
FROM Certification, Sailor, Ship
WHERE certification_type = 'Doctor';
```

Listing 2.1: An example SQL query.

Listing 2.1 gives an example of an SQL query. This query requests information about sailors who has the doctor certification type. Interpreting this query we see that it will make use of the Certification, Sailor and Ship tables. The operations included are selection for the certification constraint and joins to merge the data from the three tables. A more advanced query would also have other operations. More about these operations in Section 2.4, Query Optimization.

## 2.3 Caching

A cache is a temporary storage, and the term caching is used to describe the process of storing something that is believed to be needed in the future[18]. When using a cache, the system does not have to discard every result after they are used, but can choose to place the result in cache instead. The intention is to keep items in cache that are likely to be reused, saving the system the cost of reproducing the result.

The reason caching works is due to the locality principle[19]. The two most common appliances of the locality principle are the ones that are applicable to caching, locality in time and locality in space. Locality in time means that within a time frame a program accesses only a subset of its data. This implies that the recent history is an appropriate indication of what to cache for the recent future. A common reason for this behavior is the frequent use of loops in programs. A cache usually exploits locality in time by using, a simple yet very successful replacement strategy, Least Recently Used (LRU). Locality in space means that the data used together usually also is stored together. This is due to organization of software into modules, the way compilers allocate memory to variables and how collections of variables used together often is organized into lists, arrays and similar structures. Spatial locality is commonly exploited with prefetching. Prefetching means that the system assumes a linear access pattern and places the next element in cache while the first is still in processing. In overall, this leads to lower response times, less message traffic and better scalability of the system[20].

There are two general approaches to caching. The first is physical caching done in the form of keeping records, pages or static partitions of base tables. The other being logical caching done by keeping query results or query intermediate-results as in [9]. We will refer to the physical caching technique as data cache and logical caching as semantic cache. Data caching is the traditional caching technique. Semantic caching will be described in Section 2.3.1.

The concept of caching works well in databases. The process of producing a query result is costly, and by using cache some of these results can be fetched instantly from cache instead. In distributed database system, data is shipped from its stored location to the processing location. In a centralized database the RAM is directly above the disk in the memory hierarchy, but in a distributed database the network layer is often placed between the disk and the RAM. This in effect increases the distance between where the data is stored and where the results are produced and further increases the need for cache. The main purpose of caching in this environment is to reduce communication cost and work load of shared data sources. The most common type of cache used in databases is data cache.

### 2.3.1 Data Cache

Data cache is a term for physical cache. By physical cache we mean that the cache consists of raw data, whether it is records, pages or tables. There is no logical structure to the data. This is the most traditional caching method. Content is described in the simplest form by listing data identifiers. This creates a lot of a overhead, and leaves the cache with little flexibility to its use. It can be very effective if one is willing to sacrifice flexibility in favor of simpler maintenance of the cache. Data caching is not well suited for fine-grained caching of something other than whole units of data.

In terms of distributed databases, data cache requires a tight coupling between sites. Data caching requires similar storage and is usually coupled with shared indexes. In other words data caching is not very suitable for heterogeneous systems[21].

### 2.3.2 Semantic Cache

In the context of database systems, an entry in a semantic cache[16] is simply a result produced by a previously executed query. The entry could be the result of the complete query, or intermediate results produced during execution. As an example, a database is producing the join of three tables: A, B,

C. After producing the first join of A and B, the database would have the option of caching this intermediate result, even if the complete query has not been executed yet. Semantic cache is based on the idea of semantic locality. By semantic locality, we say that future queries is likely to be conceptually related to queries executed in the past. Instead of doing much of the same work over and over, as would be the case with related queries, there is a chance that the same result would be available in the semantic cache. One may consider semantic locality a special case of spatial locality, but then it is important to remember the distinction that in data caching the space is linear or at least euclidean, with semantic caching it is neither. This is why prefetching is not suited for semantic locality.

The semantic cache is aptly named so because of the semantic description of the data it contains[22]. The idea behind semantic descriptions is to avoid the high overhead of maintaining a list of physical pages or tuple identifiers. The semantic description offers a more elegant approach of determining what parts of a query is cached and what must be considered a remainder query. The data for the remainder query must be produced in the traditional way, while the cached query can be read directly from the cache.

### 2.3.3 Cache Hit Ratio

Cache hit ratio is a term often used to measure the effectiveness of cache. This has been the case for data cache for a long time. As data cache content is units of data ranging from whole tables to individual tuples, this is a type of cache that will be used during table scan. Any data not needing to be fetched at this point is time saved, and thus every cache hit is good for overall performance.

Semantic cache on the other hand is using another approach to caching, by caching intermediate query results and storing them with a semantic description. The nature of semantic cache makes it more likely to be used as a substitute for a more complex sub-query than a simple table scan, although this can of course also happen. It follows that the effectiveness of semantic cache is better measured in the quality of a cache entry instead of the quantity of cache entries used. The ideal cache entry is small, expensive, often used and rarely invalidated.

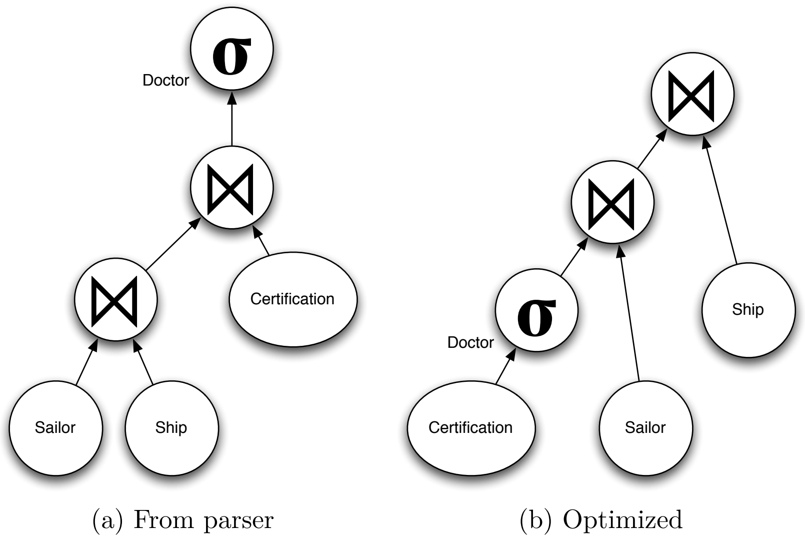


Figure 2.1: Two equivalent algebra trees for a query.

## 2.4 Query Optimization

Query optimization is the process of finding the most efficient way to run a query. The goal is to produce a query execution plan[23]. Query execution plans are often represented as an algebra tree. For those already familiar with the tree structure, the root node is the result node for the query. Each inner node is some algebra operator contributing to the result, and the leaf nodes are data sets. This can be seen in the trees in Figure 2.1a and 2.1b. A database query consists of one or more algebra operations, and these operations do not necessarily have to be executed in the same order. For instance are the trees in Figure 2.1, logically equivalent. However, their execution time can be vastly different. Most algebra operations also have options on their own on how they can be calculated and this leads to many ways of producing an execution strategy. For instance may selection operators be placed almost anywhere in the tree, but most optimizers will try to push them down so that their reducing effect will be applied as early as possible.

The query optimizer is usually seen as three components that are tasked with producing the best query execution plan[1]. The components are the search space, cost model, and search strategy. The search space is the different execution plans. Cost model is how each alternative cost is estimated.

The search strategy is how the optimizer finds the alternative with the lowest cost. The two trees in Figure 2.1 each represent a part of the search space for the query. These concepts will be further described in the next sections.

### 2.4.1 Search Space

The search space can be defined as the set of equivalent operator trees which one can reduce a single query into by using transformations[24]. Many relational algebra operations are both commutative and associative. This allows for a large search space. Some parts of the search space can however be excluded with simple heuristics, for instance it is very common to avoid Cartesian joins. Other parts can be transformed with rules that always pay off if applicable. The moving of the selection operation as far down in the tree as possible in Figure 2.1, is one example of such a rule.

Some systems allow for using the result from one operation as the operand to several operations. This changes the tree into a DAG (Directed Acyclic Graph). In systems with multi-query optimization this is essential. In single-query optimization not so much. In fact, this can only be achieved in a single query if it uses aliases, as in self-joins, or with sub-queries. Cache may also compensate for not supporting DAGs.

Most DBMSs only include algorithms for joining two relations at a time and solve this by nesting joins. This requires the optimizer to generate a binary tree of join operations. Since joins are associative, a tree of $n$ joins has $n!$ equivalents. Which one of these is the optimal choice, is evaluated with the cost model. The set of trees to consider is typically limited in two ways: how they are generated and a pruning step. The disadvantage with such limitations is that one might not find the optimal plan. A short path might lead to a longer path at the next intersection. By not looking at the total picture and not allowing oneself to change a choice once it is made, the likelihood that one will end up with a less than optimal path is very much present. Thinking ahead and considering more than one step at a time might pay off in the end.

Some classic optimizers, like System R[1], only consider left-deep trees. The class of left-deep trees, as seen in Figure 2.2, is defined as the trees where every right node is a leaf node. Figure 2.1 has two examples of left-deep trees. Left-deep trees has the advantage of being fully pipelineable [25]. A pipelined plan is advantageous with join algorithms that require reading of the entire right operand before finishing the processing of the first tuple

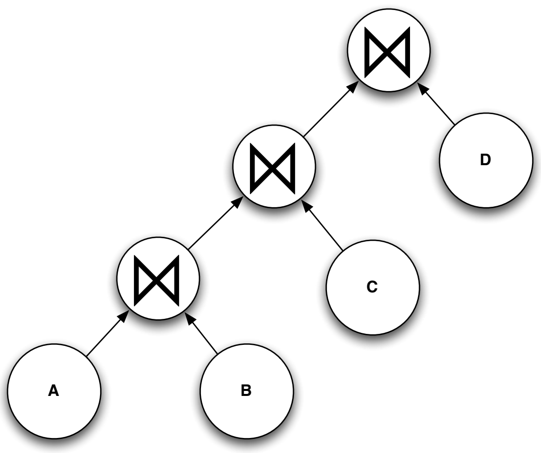


Figure 2.2: Left-deep tree.

in the left operand. Nested loops is one such join algorithm. The class of linear trees is defined as all nodes have one base relation as child, but not necessarily the right one. Hence it is a superset of the class of left-deep trees. Bushy trees, see Figure 2.3, contain nodes where both children are the result of another join. Bushy trees may on some systems be more appropriate for parallelization[26].

A system may implement more than one algorithm for computing a join for instance "hash join", "sorted join" and "index join", adding another dimension to the search space. Each algorithm typically have different properties: producing a sorted result, running faster with sorted input or good performance on very selective joins.

Join operations often have great impact on performance in relational queries, but when many relations are considered, the size of the search space can get very big[1]. Measures often need to be taken to reduce the number of operator trees, so that efficient processing is possible.

In distributed databases it is common that tables may be fragmented over several sites and each fragment may also have replicates on different sites. Figure 2.4 displays the search space for a single join in a distributed setting. The optimizer must choose which replicate to use for each fragment. However, the figure lacks one detail. The optimizer also has to choose which site to perform the join on.

In fact, every algebra operation of a query will need to have a designated

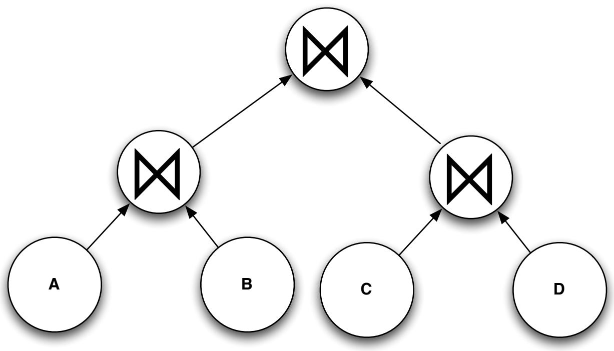


Figure 2.3: Bushy tree.

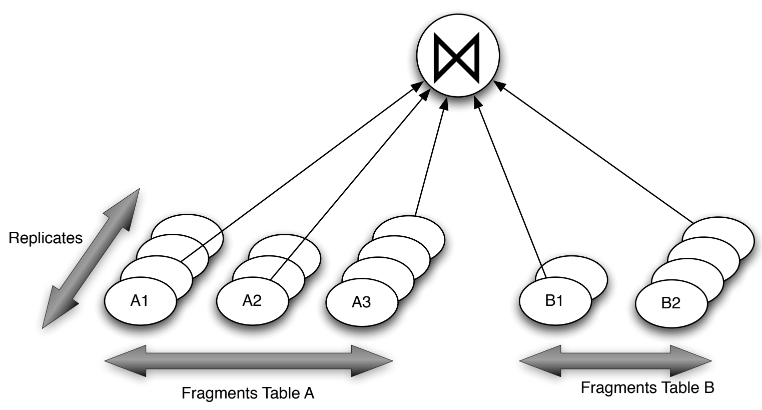


Figure 2.4: Distributed search space for a join.

site and therefore the addition of distribution increases the size of the search space by one magnitude. However, most distributed optimizers alleviate this by not considering the different local access paths to be used at each site, like index lookup compared to table scan. This can be a just simplification if the different sites with the same replicate maintain the same indexes and orderings.

Moving selection as far down in the tree as possible is still a smart approach in distributed DBMSs, perhaps now even smarter. It will reduce the amount of bytes sent and it may even allow discarding some of the fragments all together.

### 2.4.2 Cost Model

To be able to evaluate paths in query execution plans, we need a way to separate good plans from bad plans based on cost. The cost model includes the cost functions and necessary statistics to predict the cost of operators and to estimate the sizes of intermediate results. There are several ways to measure the cost of an operation. Total time, which is the sum of all components in the operation, is the most common[1]. Response time, which is the time that has passed since the query was started to a result has been produced, is another common metric. Response time is critical in real-time systems.

In distributed databases the cost model is given an extension to be able to work with the new factors introduced through distribution. The distributed cost model is in essence just a variation of the centralized cost model with a new layer of complexity added[23]. Communication cost is added because we now must take into consideration all the time and resources spent on transporting data between each participant in the distributed environment. Not only must we consider the overhead introduced with communication, but the communication cost now becomes a major part of executing a query.

$$\begin{aligned} Total\_time = {} & T_{cpu} * \#inst + T_{I/O} * \#I/Os \\ & + T_{MSG} * \#msgs + T_{TR} * \#bytes \end{aligned} \tag{2.1}$$

| | |
|---|---|
| $T_{cpu}$ | CPU instruction time |
| $\#inst$ | Number of instructions |
| $T_{I/O}$ | I/O operation time |
| $\#I/Os$ | Number of I/O operations |
| $T_{msg}$ | Time to send a message |
| $\#msgs$ | Number of messages |
| $T_{TR}$ | Time to transfer a data unit of one byte |
| $\#bytes$ | Number of bytes |

Table 2.1: Variable definitions for Formula 2.1.

Formula 2.1, with variable definitions in Table 2.1, is used to evaluate the total cost of executing a distributed query. Based on the plan used, each part in the formula will be given a cost. The sum of all these costs is the total time. Of all plans generated, the one that is evaluated to have the least total cost will be chosen to be the best query execution plan. Response time, on the other hand, discards every operation that is done in parallel and looks at the total time that has passed and not the total work done. It is important to have good knowledge about the hardware components in the system and the inter-connecting network used. Such knowledge will help make the time estimates more in line with the real world and improve the optimizer's choices.

**Processing Cost** This is the local cost representing the local resources that will need to be used to execute a query. Relevant to this cost is number of instructions and CPU speed, and number of I/O operations and disk speed. These factors is known to be important in traditional centralized databases. For distributed databases the whole situation is escalated to another scale with the cost of communication. Local processing cost will still have impact on the total cost, but the question is whether it is worth it. The case of communication cost will be described in greater detail below.

**Communication Cost** Communication cost has a tendency to dominate the work cost in a distributed database[1]. Sending and receiving data on shared networks and the Internet in particular is prone to be affected by variations in network speeds. These variations could be explained in the topology of the network or temporary high loads. For a distributed system to be able to make good decisions when doing inter-node communication, the ability to predict such variations is desirable. This is illustrated in the time between a request is sent and the time the response is received. This delay, called latency, is a good measure for response time between two given nodes.

At first glance, the simplest way would be to keep a record of every site-to-site latency in a table. Achieving good scalability with this solution is not realistic as the size of the table would grow with the square of the number of sites. Even with an abundance of storage it is not a feasible solution, as maintaining such a table up to date would be costly. If every node were to measure the latency to every other node the load on the network would be substantial as well.

A better tool for predicting inter-node latencies is synthetic coordinate systems[12]. Nodes maintain a set of synthetic coordinates that place them in a Euclidean space. The Euclidean distance between different nodes predict the latency between them. A node x need only to learn about the coordinates of another node y to determine a latency estimate. Network coordinates help to increase performance while keeping measurements overhead low in a growing distributed system where latency predictions are crucial to good decisions[27].

In the case of a system where data is replicated on several hosts, the coordinate system can be used to decide on the best replica to choose in operations. Such a replica could be chosen without the overhead of probing every node in the system for the right replica.

Vivaldi is a simple, distributed symmetric algorithm for computing synthetic coordinates[12]. In Vivaldi, each node is responsible for its own set of coordinates and to regularly exchange coordinates with other nodes in the system. Exchange of coordinates allows nodes to adjust themselves in the coordinate space according to each other. This can be seen in Figure 2.5. Nodes seek to converge against a placement that will minimize the overall error for each node. In this state the coordinate system is a good prediction tool for inter-node latencies.

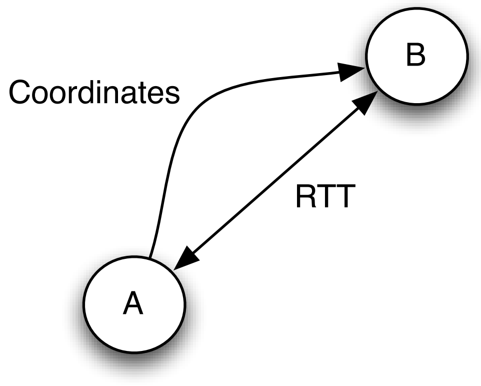


Figure 2.5: B receives coordinates from A and adjusts itself.

A large number of nodes trying to adjust their coordinates at the same time will take some time to converge. Once the majority of nodes in the system has reached their correct positions, new arrival or departures of nodes will have little impact on the given coordinates. Some disruption is possible, but the new node will not before long have found its rightful place.

### 2.4.3 Search Strategy

The search strategy is how the query execution plan is found. Without a search strategy, one would simply be evaluating pseudo-random alternatives. This is problematic for several reasons. First, one would not know when to stop. Second, one would consider options that are known to perform poorly. Finally, one would not be guaranteed to find the optimal solution, even when the search space has no local minimum. Additionally to the issues they might cause, all these cases will also contribute to more time consumption than necessary during planning.

The choice of a strategy is not as clear as the need for having one. It is usually influenced by other design choices in the system. Not all strategies are applicable in a dynamic environment where planning is done interleaved with execution of the query. A popular approach is dynamic programming. With its exhaustive search, it is guaranteed to find the optimal solution[1]. However, for queries with many relations to join, it is often considered too expensive, and probabilistic methods based on genetic programming are usually preferred[1]. Pruning is another less drastic approach to a large search space. It is the process of discarding parts that is proven, or simply considered, not to be optimal. Pruning can be done implicit as a part of the alternative generating process or with explicit filters before evaluating a generated alternative. The implicit method is inherently faster, but the explicit is more flexible as it does not require alteration of the actual algorithm. All strategies use pruning to some extent, the difference is more in whether it is actually proven that it does not contain the optimal solution or just expected not to.

Many proposals for search strategies have been made for different types of queries and most commercial DBMS's implement several of them. However, the exact details of how they are utilized often remain secret as this is one of their main competitive areas. We looked into different strategies in [10] when we made the solution for the DASCOSA optimizer.

## 2.5 Using Cache

In this section we will address the implications of cache to the optimizer in a DBMS. The optimizer may be oblivious to the cache and create plans as if there were no cache. This is usually the case with data caching. Then the cache hit is not detected until the plan is in execution. We refer to this as implicit caching.

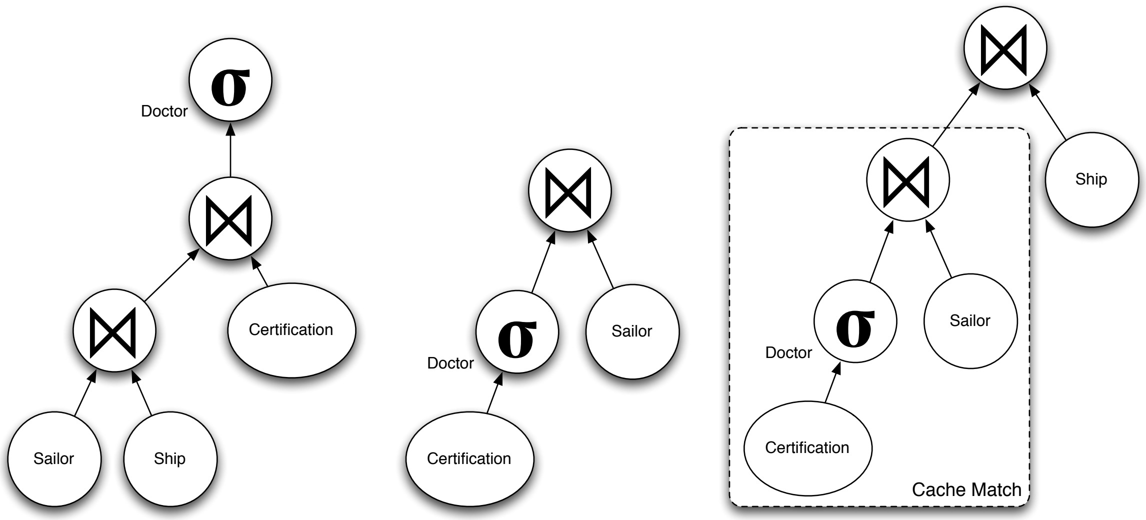
Implicit caching might give a low cache hit rate with a semantic cache. For example, the cache contains the join of tables A and B and the optimizer is tasked with finding the best plan for joining A, B and C. If the best plan without cache is to first join tables B and C and then join with table A, then that is the plan chosen with implicit caching. Clearly such a plan will not give a hit for the cache entry with A and B. If the optimizer is aware of the cache's contents and is capable of planning for cache usage by inserting cache entries at the planning stage then it has what we refer to as explicit cache usage.

Cache in general is used to make data access faster, but it is not given that it will give a speedup in all situations. This is not just an issue in terms of making optimizers cache aware. The optimizer has to do the same considerations for materialized views[15]. A materialized view means that the system maintains an up to date copy of a query's result set on disk. Using a materialized view or cache entry might exclude using an advantageous index, join ordering or site. For the query in question the optimizer has to choose between recalculating the required parts of the result or using the materialized copy, but before deciding upon the cost of using a cache entry a cache aware optimizer must decide which entries are applicable. We will address this issue in the following section.

### 2.5.1 Query Matching

Query matching is a term used for the problem of detecting if a cache is useful to a query[28]. If query and cache entry is represented as a relational algebra tree, then query matching can be done through transformations on the query tree so that the entire query tree or one of its subtrees equals the cache tree. Then parts of the query tree may be replaced with one or more cache trees, without altering the result. Because two algebra trees can vary in structure but still produce the same result, the real challenge lies in detecting equivalence.

Figure 2.6a displays a typical query tree as generated by the parser. This is the same query as depicted in Listing 2.1. Figure 2.6b displays what is cached in the query. Should one execute the query as represented in Figure 2.6a the executor would not be able to utilize the cache. This is because the cache entry does not match any of the partial results. In order to do so the



(a) Original query (b) Cache entry (c) Query transformed

Figure 2.6: Example query match transformation.

query tree must be transformed before execution. Figure 2.6c gives a tree that is both equivalent with the query and contains the cache as a subtree.

As was seen in Section 2.4 this query is transformed into the query in Figure 2.1b before execution. Evidently the cache only caches the queries that are executed and hence only queries that have been optimized. In terms of cache utilization this has both pros and cons. The advantage is that due to the optimizer being deterministic it will increase the similarities in the executed queries. The optimizers' tendency to apply reducing operations early may result in cache entries with so specialized constraints that they are highly unlikely applicable for future queries. Generalizing query results before submitting them to cache is a suggested solution for dealing with over-specialized queries[29].

In situations where it is worth while optimizing heavily on transport cost, it could be useful executing a remainder query. A remainder query is a query that fetches the tuples a cache entry lacks in order to be useful for the query in planning. For instance, if the cache entry contains all tuples with id more than 500 and the query needs all tuples with id less than 1000, one could execute the query constraint anded with the negation of the cache constraint as a remainder query. One would then have to execute the query's constraint on the cache entry and union the result with the remainder query. However there are situations where the remainder query is just as expensive as the actual query. For instance the site executing the remainder query might have to do a scan of the entire table irregardless of the constraints, and the savings in transfer cost executing the remainder query might not match the work of the extra local processing. In order to consider remainder queries an optimizer should have a fairly accurate cost model.

When the database through query matching detects that there is data cached that that might be useful to a query, measures must be taken so that the cache is utilized in the best way. Query rewriting is such a technique, which seeks to change the structure of the query so that the old result in the cache can replace some or all of the current query. Good query matching plays an important role here, so that one can be sure that the final result does not contain wrong information.

Query folding[30] is an example of a query rewriting technique. It focuses on determining whether a query can be answered with the resources at hand. These resources could be materialized views, cached results of previous queries and remainder queries. Query containment is a special case of query folding, in which a query is checked to see if it can be answered using another query. If that is the case, it is said to be contained.

## 2.6 Cache Investment

Cache investment[9] is a technique for combining data placement and query optimization in a manner that does not change the query optimizer directly. Cache investment affects data placement by influencing the optimizer to make suboptimal choices regarding operator site selection. These suboptimal choices will be based on policies producing cached data placement beneficial for subsequent queries. Because this technique only influences the optimizer, it is always up to the optimizer to determine if a suboptimal choice is worthwhile. The key selling point of this technique is the effective integration with an already existing query optimizer, such as a cost-based query optimizer like R*[11]. This is shown in Figure 2.7. Cache Investment as described is designed for a distributed DBMS with a server-client architecture, but there is nothing in the concept that disallows it for operating with a more flexible peer-to-peer (P2P) architecture. On the other hand, it would have to be adapted to its new environment.

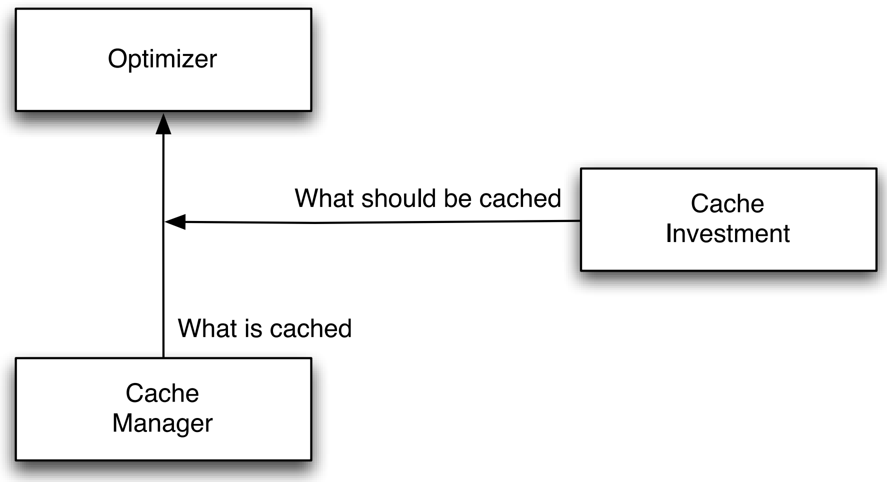


Figure 2.7: Cache investment[9].

As an example, the cache investment based on some kind of policy decided that some part of a query would be a good idea to cache. The cache investment modules influences the query optimizer by telling it that such a cached result in fact exists on a given site. This might be true or not. The cache investment module is in fact allowed to provide the query optimizer with false information about a cache. It is important to note that this is not going to be some malicious lie, but rather a friendly nudge telling the query optimizer that keeping such data in the cache would be a good idea. The cache investment knows this because of the policy it is using to keep track of data usage. It is ultimately up to the query optimizer to decide if it should believe the cache investment module, based on its own calculations on execution cost. If the query optimizer decides to go through with this fictional cache and produce the result, the data can be cached and the cache will become reality.

Here is an example on how cache investment works. Consider three sites each with one table A, B, and C. The cache investment identifies the result of join of A and B as a profitable cache at site 2, since A and B are frequently used together. When a subsequent query consisting of the join of all three tables is submitted to the database, the cache investment module informs the optimizer that the join of A and B exists on site 2. The optimizer evaluates a plan consisting of the false cache on site 2 and the retrieval of table C from site 3. The optimizer determines that this is the best plan and sends it along to be executed. While the cache does not exist, the cache will have to be created during execution this first time. This might hurt performance during the first run, but any subsequent queries will now profit. Since the cache is based on statistics provided by the cache investment module, we have better assurance that this cache should prove to be useful in the future.

Cache investment is not a technique for the actual caching process, but more like a helpful tool for bringing data together to produce a good candidate for caching[9]. Data replacement in the cache is left to policies native to the cache being used, such as the LRU-policy.

### 2.6.1 Identifying Caching Candidates

The purpose of the cache investment policy is to determine what data that is beneficial to cache. Such a data item is known as a candidate in the context of cache investment[9]. In the process of determined good cache candidates, there are two useful terms: investment cost and return on investment. Investment cost is the sum of all work required to put a data item into the cache. This work will only pay off if the cached data item will be used in any subsequent queries. Return on investment (ROI) is the value for what can be gained by caching the data. Finding good candidates for caching is a trade-off between investment cost and return on investment. How this is calculated is left for the policy being used by the cache investment module. Below are two policies suggested in the paper by Donald Kossmann[9] on how to identify good caching candidates.

**Reference-Counting Policy** The idea of Reference-Counting is to count the number of queries in which a table is used[9]. With knowledge about the popularity of data this policy is able to influence the query optimizer. This approach does not calculate ROI for cache investment, but it does try to maximize the value of every unit of space used in cache. This is the knapsack problem based on value relative to size for each table. The technique requires intimate knowledge about the cache size and free space on all sites. In a distributed setting this level of knowledge can be hard to achieve.

**Profitable Policy** The Profitable policy is directly trying to estimate what the cache investment will cost and what the gain would be in form of ROI.

The technique makes active use of the query optimizer to compute the cost of the cache investment, even before the cache candidate information is used to influence the query optimizer for the actual query. ROI is computed as the cost of a query without cache minus the cost of the query with cache. This estimate can tell us about the cost savings of doing this cache investment. The criteria for suggesting this cache candidate for a given query is that the candidate must be contained in the current query, the ROI must be greater than the cache investment cost and its value in cache must qualify according to the Reference-Counting policy described above.

## 2.7 DASCOSA

DASCOSA is a distributed P2P database. It will serve as our platform for doing implementation and evaluation of our solution. DASCOSA is being developed at Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) for the purpose of doing research into distributed DBMS. DASCOSA as a system consists of a number of sites where each site contains data and participates in the query processing. Each site is autonomous and responsible for maintaining its own data and contributing to the system index. DASCOSA stores its index in a distributed hash table (DHT). The DHT index is responsible for the collective meta-data and other states that affect the whole system, as seen in Figure 2.8. The DHT is also responsible for connecting the sites and routing traffic between them. DASCOSA is implemented to use FreePastry[31], which is an open source DHT. The use of P2P technology in DASCOSA seek to achieve better scalability and availability.

### 2.7.1 Architecture Overview

The architecture of DASCOSA is illustrated in Figure 2.9. Its main components are: query processor, storage and DHT. The query processor is responsible for handling queries and producing a result. The storage component manages all data locally stored on a site. The DHT connects all the sites and handles communication in the form of lookups.

The query processor handles queries. To be able to do this it is divided into three parts: parser, planner and executor. The parser takes an SQL query and transforms it into a relational algebra tree. The planner transforms

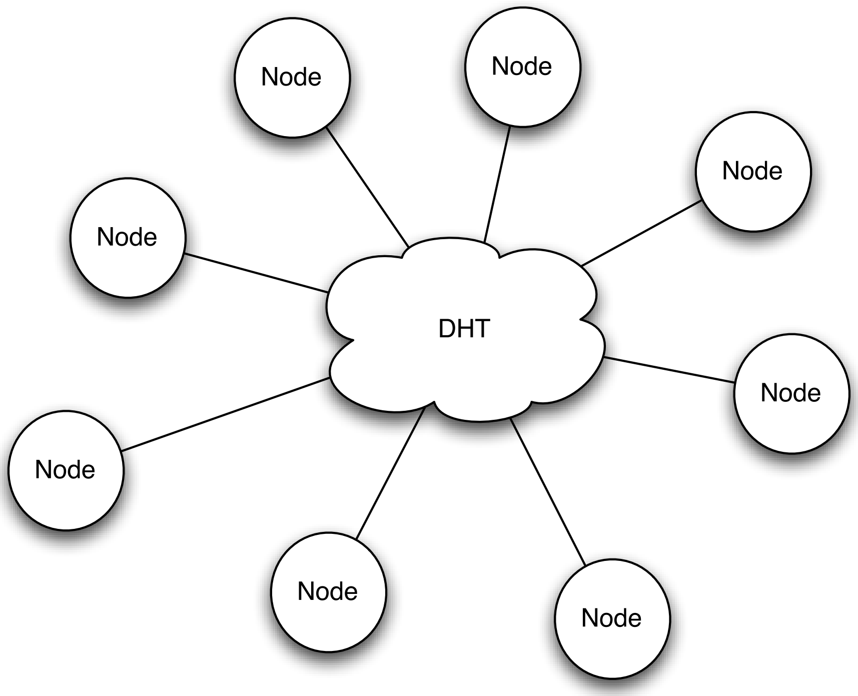


Figure 2.8: Dascosa DB with sites connected through the DHT.

the algebra tree further by enumerating alternative plans and evaluate them. The executor uses the best plan found by the planner and executes it to produce the result.

Node allocation to sites in DASCOSA is deterministic. For a given set of coordinates, operations are guaranteed to end up at the same site. DASCOSA starts by letting the planner find the plan with the cheapest transfer cost. In order to do this the planner evaluates different replicates, join enumerations and sites for each join in a dynamic programming fashion. Then it proceeds by assigning sites to parent nodes of the final join, always choosing to assign a parent node to the site of the leftmost child. This deterministic behaviour is advantageous for implicit cache usage, more on this in Section 2.7.4.

The storage component manages data stored locally. It handles the local database and meta-data about local table fragments. The storage component is also responsible for maintaining its share of the distributed index. DAS-

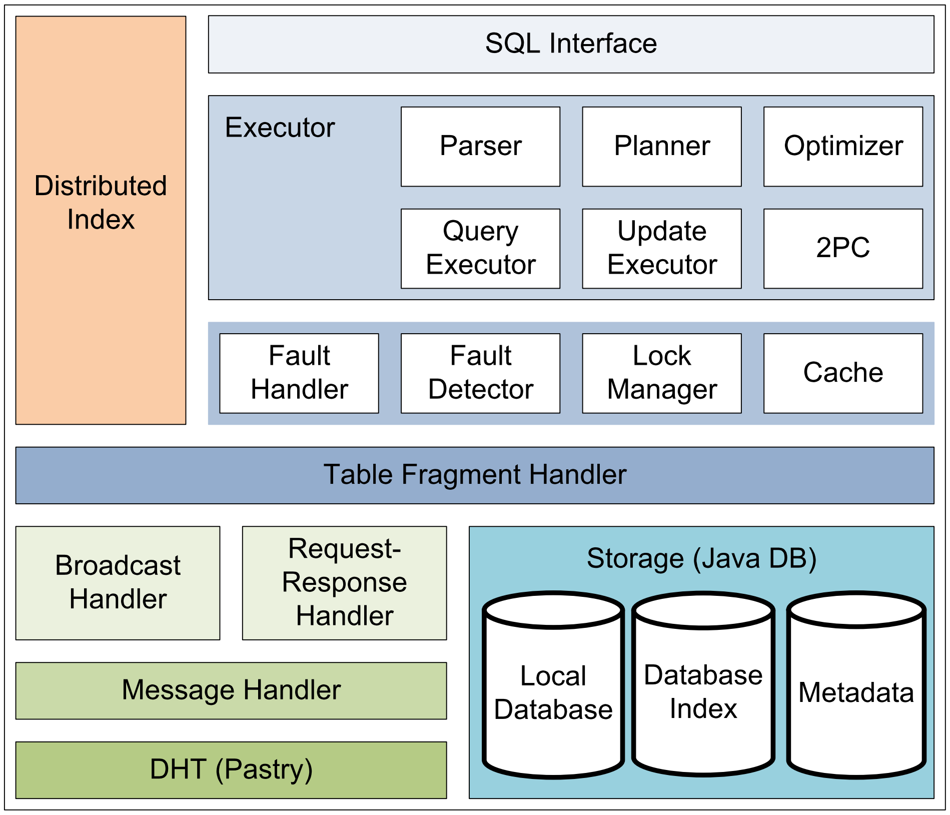


Figure 2.9: The DASCOSA architecture[32].

COSA is currently using Derby DBMS[33] for persistence within the storage component.

The DHT is responsible for loosely connecting each site in the DASCOSA network. Lookups are done through the DHT to locate the table fragments needed for the query. The DHT is used for all communication of administrative purposes. For transfer of data sets a direct link between sites is preferred to ensure the best throughput possible. This way DASCOSA also avoid clogging the channel for messages that contribute to keep the distributed system in a consistent state. The DHT in DASCOSA is implemented using FreePastry[31]. FreePastry enables lookup and message routing without every node being aware of one another. This is an important property to handle churn (sites leaving and joining) in a large network.

### 2.7.2 Query Processing in DASCOSA

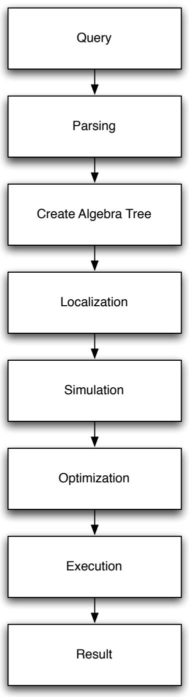


Figure 2.10: Query processing [32].

Figure 2.10 shows the stages of query processing done in DASCOSA. This closely resembles the traditional way of doing distributed query processing as detailed in [32]. The query undergoes several stages: parsing, algebra tree creation, localization, simulation, optimization, execution and finally result processing.

Parsing is the stage where the query SQL is put through syntax verification and broken down into operations. These operations are assigned to algebra nodes in an algebra tree. DASCOSA's parser builds the algebra tree as a left-deep tree as explained in Section 2.4.1.

The Localization stage collects information about which sites that are relevant for the query based on the table definitions from the index. Every site holding a table fragment of a table involved in the query is included. Information about candidate sites is found in the distributed index. A simulation is done to make sure that all the required information is in place.

Plans are generated during the optimization stage. The set of plans generated is known as the search space, as explained in Section 2.4.1. The optimizer will be further explained in Section 2.7.4.

Execution of the query happens in the last stage. The subtrees of the distributed query plan are assigned to sites according to the findings of the query optimizer in the optimization stage.

### 2.7.3 Query Optimization in DASCOSA

Our previous work on the DASCOSA query optimizer is described in [10]. The optimizer is a cost-based optimizer inspired by the well-known distributed query optimizer R*[11]. Other optimizer concepts was also analyzed during the design of the optimizer, such as Mariposa[34], Distributed INGRES[1] and SDD-1[35]. The optimizer takes an exhaustive approach to determining the best query execution plan by applying a bottom-up dynamic programming algorithm to the search space. This search strategy determines the best query execution plan and optimize heavily on communication cost. Synthetic network coordinates is used to give a representation of each site's distance to each other in form of latency. This information is used to communicate with the nearest nodes with the best response time. This provides DASCOSA with the ability to do "smart" site selection during planning.

Table 2.2 shows how the DASCOSA query optimizer evaluates the cost for each plan. Each table cell symbolize the calculated cost of producing the join at the given site. All alternatives are evaluated. After evaluation the options with the lowest cost are chosen, as seen in Table 2.2. The plan chosen in this example would be the join of A and B first, then join the result with C and finally with D. The final plan is not known until the complete plan has been evaluated.

| Step | Tables | $Site_0$ | $Site_1$ | $Site_2$ | $Site_3$ | $Site_4$ |
|---|---|---|---|---|---|---|
| 1 | AB | x | | | | |
| 1 | BC | | | | | |
| 1 | CD | | | | | |
| 2 | ABC | | | | x | |
| 2 | BCD | | | | | |
| 3 | ABCD | | | | x | |

Table 2.2: Query optimizer.

### 2.7.4 Caching in DASCOSA

The existing caching solution for DASCOSA[36] is based on semantic caching. The algebra node tree structure plays a role in the caching process. During execution, the result of every algebra node is considered as a candidate for caching. How an intermediate result is weighted for caching is based on certain user-defined parameters, but for the purpose of this project we will stick to simple LRU.

Cached content does not play an active role during query planning. Instead, during execution, a cached result can be substituted for an algebra node and sub-tree. This is only if a match can be found between the cached content and the result that would be produced by the algebra node. In other words, DASCOSA has implicit cache usage as described in Section 2.5. Implicit cache usage with a semantic cache in a distributed DBMS requires that the same sub-plan is planned for the same site repeatedly, possibly by optimizers at different sites.

The caching module in DASCOSA continuously considers new candidates for caching during execution. Data replacement in cache is done with the use of a Least Recently Used (LRU) queue. That means that an algebra node, if identified as a suitable candidate for caching, replaces the least recently used element in the queue.

Although the optimizer is currently oblivious to the cache, the cache module does advertise its contents. At each site it publishes the cache entries in a similar fashion as table fragments are published. The difference is that table fragments only relate to one table while cache entries may contain more than one table. This is solved by a deterministic hashing function that decides which of the tables to use for choosing an index site. This allows for piggybacking pointers to cache entries when the initiating site does its ordinary table lookup. As the initiating site has to lookup all the tables in the query, it does not matter if the cache entry is only indexed at one site. The entries are identified by a string representation of the algebra trees they are generated by. So far this has only been utilized to do simple exact matches of query trees in performance tests of the cache module without the optimizer by using preplanned hard coded queries. In other words it is not capable of doing query transformations like suggested in Section 2.5.1.

# Chapter 3

# Design and Implementation

This chapter will describe the work on our solution to enhance caching in DASCOSA. The work will be split into three phases, each focusing on a particular extension to the DASCOSA database and its query optimizer, which we developed prior to this project[10]. Each phase will have a design section and an implementation section. The design section will explain how we approached the problem and discuss alternatives for solving the task. The implementation section will explain how we carried out our design from the preceding section. When a problem with the design is encountered, this will be discussed and we will explain how the problem was dealt with.

The three phases that can found in this chapter are:

1. Making the Optimizer Cache-Aware

2. Extending Operator Support

3. Cache Investment

## 3.1 Overview

The goal is to integrate the concept of cache investment with the semantic cache and peer-to-peer (P2P) nature of DASCOSA. This will be achieved through three phases, each extending DASCOSA and its query optimizer with new functionality. The first phase will make the query optimizer cache-aware, that is make it able to plan for cache utilization. By letting cache play an active role during planning we seek to achieve a much better cache usage. The challenge here is query representation and query matching when dealing with semantic cache.

The second phase will extend the query optimizer to support selection and projection. This is required for DASCOSA to support a broader range of queries. These queries will be more applicable for testing the profit of cache investment.

The third phase is the actual cache investment. We will address issues such as query logging, history analysis, cache candidate identification and site selection to do cache investment in the context of semantic cache.

## 3.2 Making the Optimizer Cache-Aware

The goal of this phase was to make the existing optimizer in DASCOSA capable of planning for cache usage. In other words to go from implicit to explicit cache usage, as described in Section 2.5. As the existing optimizer only considers joins and replicates, this should be a feasible initial goal and we expect it to be advantageous to have this in place before extending the optimizers query transformation capabilities with more algebra operations. The motivation for making the optimizer cache-aware can be described with the following example. Given two sites located relatively close, for instance on the same local area network, both sites executing the same two queries repeatedly, but in an interleaving fashion, as given in Table 3.1, and the tables are stored at a remote site. If in such a situation the cache cannot contain both table A and B and the cache uses Least Recently Used (LRU) as its replacement strategy there will be a thrashing situation and the stored entries will never be used. If the optimizers at both sites where aware of one another's cache entries then at time step 2 they would simply retrieve the data from each other instead of the remote site.

| **Timestep** | $Site_1$ | $Site_2$ |
|---|---|---|
| 1 | $Q_A$ | $Q_B$ |
| 2 | $Q_B$ | $Q_A$ |
| 3 | $Q_A$ | $Q_B$ |
| 4 | $Q_B$ | $Q_A$ |
| 5 | $Q_A$ | $Q_B$ |
| 6 | $Q_B$ | $Q_A$ |

Table 3.1: $Q_A$ scans table A and $Q_B$ scans table B.

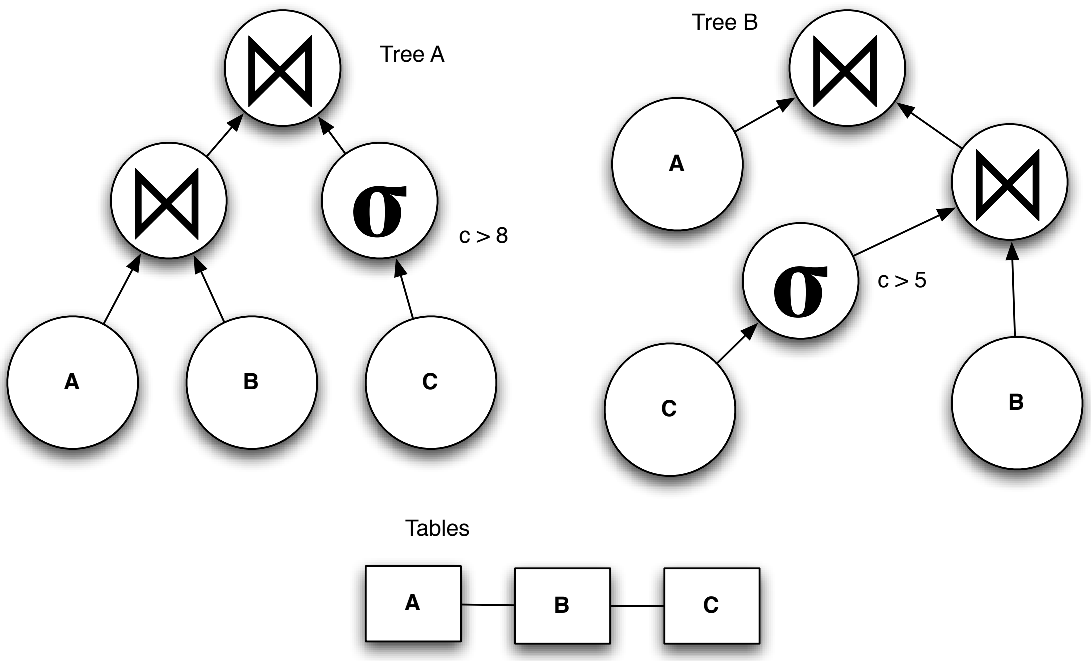


Figure 3.1: Two comparable query trees.

### 3.2.1 Design

The goal is to make the optimizer cache-aware, which means letting it know about cached data and actively include this information during planning. The solution will be done in 3 steps. The first step is query matching. Query matching is the term used for the process of determining if a cache entry is relevant for a given query, as detailed in Section 2.5.1. The challenge lies in finding if two queries with their relational algebra trees, which can be very different in structure, have any subtrees in common.

The second step, planning with cache, describes how the actual planning process must be changed when a new type of elements, cache entries, are added to the search space. The focus on the third step, cache in execution, will describe how cache nodes should be handled when the query execution plan is used and what should happen if the cache entry planned for goes missing after the plan has been produced.

**Query Matching** As described in Section 2.5.1, finding a match between a query and the contents of a cache entry can be a challenge. Often in a potential match, the cache entry resembles some or all of the query but with minor deviations. To discover that there is a match, these small differences will need to be worked around. These are usually tree structures with the same result, but in which the nodes are arranged differently. Clearly we needed some form of representation that would simplify this matching. We considered two different approaches: normalizing the tree structure and removing the structure altogether. Tree normalization is a thorough and solid technique that would preserve the structure. This approach with the structure intact should be more flexible in terms of supported queries. On the other hand, the nature of relational algebra and placement of operators gives us many trees that are comparable but has structural variations. An example can be seen in Figure 3.1, where Tree A and Tree B have different orderings of the nodes. If their constraint values was equal they would produce the same result. Determining when and if these would be equal is a challenge. Tree normalization in this case would work, but drastic changes would be needed to match the trees while ensuring they still produce the same result. Removing the original structure altogether and finding a new representation might be more effective for a specialized problem. We choose to pursue this path when designing query matching in the optimizer. We consider tree normalization to be cumbersome and time-consuming and outside the scope of this project.

Therefore in our approach we choose to look into so called PJS-queries[28], that is queries which are limited to only containing projection, join and selection. In such a query the most important elements are the tables involved, the constraints set by the selection-operator, and the attributes determined by the Projection. The ordering of algebra operations is not important, because any order will produce the same result. Although there are some join operations where order plays a role, we assume that only natural joins are used in our context.[1] To represent such a result set we introduce a Result Set Identifier (RSID). The RSID describes the content of the result set, instead of the operations producing it. This way it is easier to overlook variations in

[1]Equijoins would certainly be more flexible, but they are not necessary to do cache investment and the DASCOSA optimizer does not support them. However the work done in this report should easily apply to equijoins as well. For instance could the join attributes be added explicitly as an extension to the RSID or implicitly by making them part of the selection constraints.

the sub-tree in cases where the result produced is the same. This structure is similar to node labels introduced in[29].

This query can be represented as the following RSID:

```
SELECT c_name
FROM nation, region
WHERE r_name = "Africa;"
```

| | |
|---|---|
| **R:** | nation, region |
| $\sigma$**:** | r_name = "Africa" |
| $\pi$**:** | c_name |

When matching a query and a cache entry using RSID, the relations, constraints and attributes of both query and cache is compared. During this comparison containment is checked. As the optimizer up until now just deals with joins only the table set of the RSID is of interest for this phase. In this phase, containment between two RSID is defined as given in equation 3.1.

$$RSID_A \supseteq RSID_B \Leftrightarrow Tables_A \supseteq Tables_B \tag{3.1}$$

**Planning with Cache** When creating a query execution plan DASCOSA uses a dynamic programming algorithm based on R* from [11]. During the base step the planner is tasked with finding a plan for each table in the query. Each additional step creates plans of incrementing size by joining two previous plans. In the planner these basic building blocks for plans are called options. An option represents either a join or a table scan and the site to execute it. Options representing joins also has pointers to the options producing the operands.

To let the optimizer be able to utilize results in cache we introduce a new option called a cache option. The cache option will contain information about the cache entry that will let the optimizer evaluate it on the same level as the normal options for plans. Like described earlier, the optimizer use a dynamic programming approach to find the best query execution plan. This is done by joining two previous plans creating a new and larger plan. While step-wise building a plan this way, the best plan can be decided when all options in the search space has been exhausted. Introducing cache options provides the optimizer with new alternatives that will produce plans with cache. The cost model differentiates the cache options in such a way that they may be chosen when profitable to the plan as a whole.

| Tables | $Site_0$ | $Site_1$ | $Site_2$ | $Site_3$ | $Site_4$ |
|---|---|---|---|---|---|
| AB | x | | | | |
| BC | | | | | |
| CD | | | | Cache | |
| ABC | | Cache | | x | |
| BCD | | | | | |
| ABCD | | x | | | |

Table 3.2: Query optimization with cache.

Table 3.2 shows how the query optimizer works with cache entries during planning. The $x$ marks the sites that would originally be chosen for the plan. In this case there are cache entries on $Site_3$ for the join of C and D, and $Site_1$ has a cache entry for the join of A, B and C. The optimizer can now choose to substitute any cache with a matching sub-plan if this results in a lower cost. In this case it might be better to use the cache entry on $Site_1$ and then join this cache entry with table D to complete the plan.

**Cache in Execution** As mentioned in Section 2.7.4 the existing caching module in DASCOSA only inserts cache nodes when the site executing an algebra node recognizes that the executing node has an exact match with a cache entry. We refer to this as implicit cache usage in contrast to the extensions made to the optimizer in this phase where the optimizer may explicitly plan for cache usage by inserting cache nodes at the planning stage.

Explicit cache usage runs the risk of cache misses, as the decision to use cache is done before the node arrives at the site with the cache. In order for DASCOSA to handle cache misses, the optimizer augments the cache nodes with an alternative plan. The alternative plan is generated from the best non-cache option similar to the cache option.

Cache in nature continuously runs the risk of being replaced by other data, therefore we had no guarantees that the cache would exist long enough to be there when needed. This was no problem before our proposed solution, because cache was left out of planning. With this new approach that made the query optimizer aware of all the options provided by the cache, we needed a way to handle this special case that might arise.

Initially we had three options:

1. Abort and restart
2. Pause execution and create cache at given site
3. Fall back to alternate plan for the cache node

We do not consider aborting query execution when a cache miss occurs as an acceptable strategy, although DASCOSA has support for partial query restart[37]. Of the three, the last was chosen. Because of the exhaustive search strategy of the query optimizer in DASCOSA, an alternate plan was already found. Integrating this to our solution was just a matter of adapting the cache node by allowing it to have an operand. This operand would be the top node of the alternate plan, and would be executed in case the cache entry for the cache node was not found. We will see later how this helps us to do cache investment.

### 3.2.2 Implementation

To successfully do query matching, we needed an alternative to the query string that identified cache entries. Our solution was the introduction of data representation called RSID, which is explained above. When a cache entry was created, an RSID would also be created and published to the index. Just as the string representation was in Section 2.7.4. Once the RSID for each cache entry was available in the index, it could be retrieved through normal index lookups. During query processing in DASCOSA, this is called Localization and is explained in Section 2.7.2.

As the optimizer in DASCOSA at this stage only considered join orderings, only the relation part of RSIDs are used. This limited the potential number of cache entries, but in the near future we plan to change this. With potentially many different cache entries, the search space could become quite large. Clearly there was a need to filter out the inapplicable cache entries before exposing them to the actual query optimization algorithm. We chose to do this at the indexing site, by adding the query RSID to the ordinary table lookups. This way only RSIDs of interest to the optimizer was sent back as a part of the index lookup. This implied that containment checking was done on every site were the index was checked for data involved in the query. Still there was no duplicate work done as each of the indexing sites had pointers to different cache entries. By filtering out RSIDs that was not applicable for the query, we ensured that the search space as well as the message sizes was not extended unnecessary. Another advantage by colocating the containment checks was that the indexing site's storage could be organized in a manner that allowed for discarding part of the entries.

To make the optimizer become aware of the cache, we took advantage of its exhaustive nature. The optimizer attempted to produce the best execution plan using the options given by the search space. These options could be either table scans or algebra operations. By adding the cache entries to the search space as cache options, the optimizer could evaluate cached data on the same level as table scans and algebra operations. The query execution plan would be produced in the same fashion, but with potential cached data included right from the start.

## 3.3 Extending Operator Support

The goal of this phase is to add optimizer support for the selection and project operators which make the query optimizer able to process the whole range of PJS-queries. The selection operator would require the planning step for the optimizer to be compatible with constraints and how these will change the data size exchanged by the sites. When dealing with the selection and project operators it is important to pay attention to their placement in the algebra tree. Both operators remove elements from the data set (selection horizontally and project vertically), and if some element is removed at the wrong point, it will have consequences for the rest of the execution.

### 3.3.1 Design

The first step towards realizing selection support in the query optimizer is to add constraint information to the RSID and the basic building blocks for the optimizer, the options. We now consider phase 1, described in Section 3.2, to an integral part of DASCOSA and will design the operators in such a way that they support semantic cache.

To determine if a cache RSID will be usable, it will be checked for containment against the query RSID. By containment we mean that every element of the cache RSID must exist, or be contained, in the query RSID. In the previous phase this simply required the tables of the cache RSID to be a

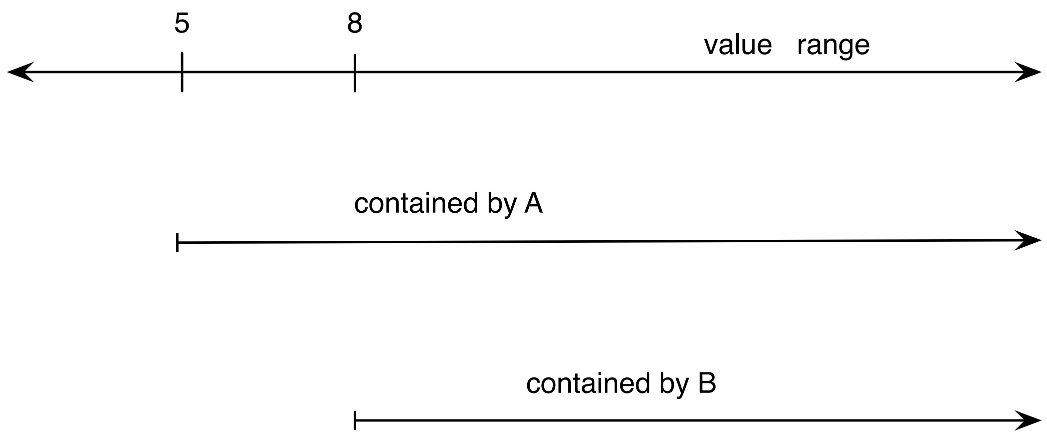


Figure 3.2: RSID containment.

subset of the tables in the query RSID, as given in Equation 3.1. This rule is still relevant, but it will have to be extended. We will now refer to this rule as table containment as given in Equation 3.2.

$$tableCon(RSID_A, RSID_B) \Leftrightarrow Tables_A \supseteq Tables_B \tag{3.2}$$

In addition to table containment, we have to consider selections and projections. So we extend Equation 3.1 to become Equation 3.3.

$$\begin{aligned} RSID_A \supseteq RSID_B \Leftrightarrow & tableCon(RSID_A, RSID_B) \\ & \wedge projectionCon(RSID_A, RSID_B) \\ & \wedge selectionCon(RSID_A, RSID_B) \end{aligned} \tag{3.3}$$

The projection containment is not so different from table containment, except it is the other way around. If a query is to use a cache entry, all the columns it requires have to be included for all the tables in the query. Extra columns in the cache entry is no problem. Then we get Equation 3.4.

$$\begin{aligned} projectionCon(RSID_A, RSID_B) \Leftrightarrow \forall p[(p \in AllColumns(R_B) \\ \wedge p \in \pi_A) \rightarrow p \in \pi_B] \end{aligned} \tag{3.4}$$

We define selection containment as that all selection constraints in the cache entry has to be part of the query or be contained by a constraint in the query and we get Equation 3.5.

$$\begin{aligned} selectionCon(RSID_A, RSID_B) \Leftrightarrow \forall y_B \exists y_A [y_B \in (\sigma_B - \sigma_A) \\ \wedge\, y_A \in \sigma_A \wedge contained(y_B, y_A)] \end{aligned} \quad (3.5)$$

An example of constraint containment can be seen in Figure 3.2, where the value range for B is contained in A. This is the value ranges for the query represented by Tree A and B in Figure 3.1. From Figure 3.2 it is clear that the constraint B can be applied either to the entire table or on the result of constraint A. In order to adhere with previous terminology we would like to say that constraint B contains constraint A. This is why we define the contained relation between two constraints by the tuples they exclude, as given by Equation 3.6.

$$constraintCon(x, y) \Leftrightarrow \forall t_x t_y excludedby(t_y, y) \rightarrow excludedby(t_x, x) \quad (3.6)$$

In other words the containee does not exclude any tuple not excluded by the container. In which case it would also be applicable for the given query as the container can be applied after the containee and the result would be exactly the same as if only the container had been executed.

Table 3.3 shows which logical operators which can be substituted in the containment technique used for matching queries against cached data. From the table we can read if a constraint is contained within another constraint, and if necessary, what data is not contained and must be fetched independently. A contained query must satisfy Equation 3.6 to qualify. The process of generating such remainder queries and evaluating if they are profitable is outside the scope of this report. Although we do not see any immediate problems with doing so, our time frame is limited and remainder queries are not necessary to do cache investment.

| **A** | = | **!=** | < | <= | > | >= |
|---|---|---|---|---|---|---|
| 1 | | xy | xy | xy | | |
| 2 | | xy | xy | xy | | |
| 3 | x | y | y | xy | | x |
| 4 | y | x | | y | x | xy |
| 5 | | xy | | | xy | xy |

Table 3.3: Values of A in relation to x = 3 and y = 4.

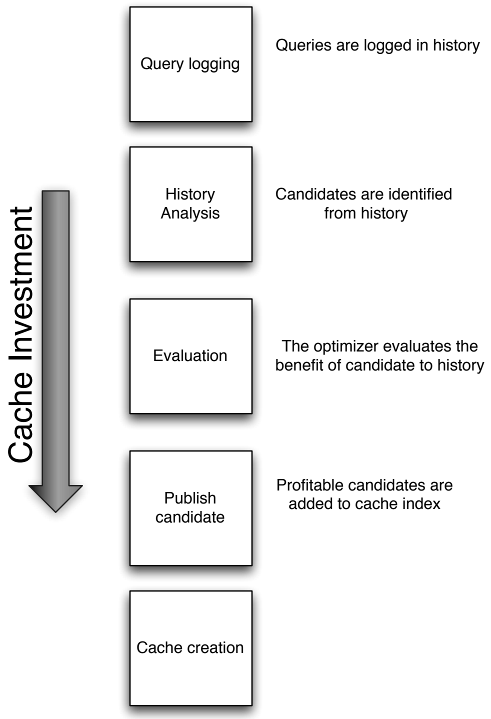

Query logging
Queries are logged in history
History Analysis
Candidates are identified from history
Cache Investment
Evaluation
The optimizer evaluates the benefit of candidate to history
Publish candidate
Profitable candidates are added to cache index
Cache creation

Figure 3.3: The process of cache investement.

### 3.3.2 Implementation

The implementation of selection support was done by adding a set of constraints to the options used in query planning. With the new set of constraints the planner was able to determine a reduction estimate for join operations. As explained in the design section above, the constraints on each option was made into independent constraint nodes when building the query tree from the query execution plan produced in the optimizer. These nodes was placed directly above the algebra node linked to the option with constraint. Except for constraints on table scans which was performed by Derby before the data was loaded into DASCOSA.

Because of time constraints and low priority, the project operator was not a part of the solution at this point. This will be a part of future work on the DASCOSA optimizer.

After the best plan has been determined, it must be translated into an executable algebra tree. The plan will be traversed top-down. When an

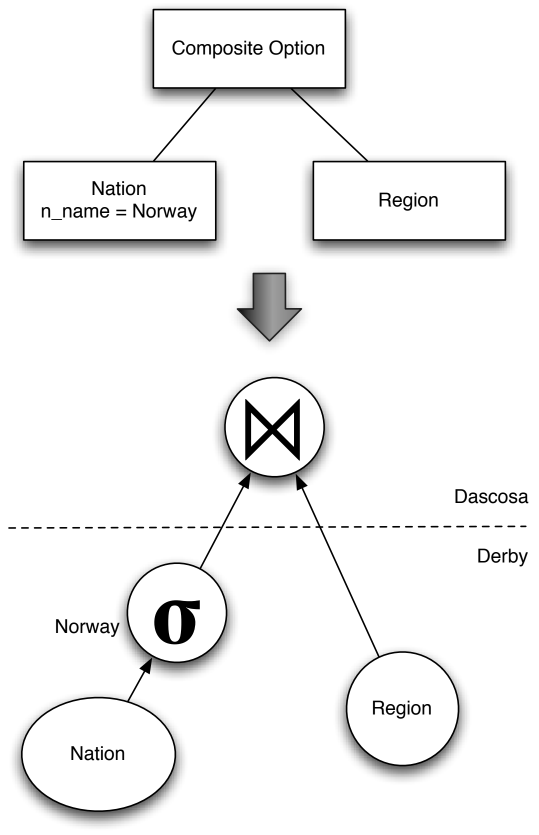


Figure 3.4: A plan with constraints from the query optimizer is translated into an algebra tree.

option with constraints is encountered, two nodes will be created. First a constraint node with the given constraint is created, then the algebra node for the option itself is created as a child node of the constraint. This way, the constraint will be executed immediately over the algebra node, as seen in Figure 3.4.

Constraints on leaf options, or table access operators, will be handled as a special case. The constraint will in this case by applied directly on the table access in the database layer under DASCOSA. This is a highly efficient way of reducing the data set before it enters DASCOSA.

## 3.4 Semantic Cache Investment

This section will describe our solution for how semantic cache investment is applied to the P2P structure of DASCOSA and its existing semantic cache.

### 3.4.1 Design

The design for our adaption of cache investment consists of 5 steps, as seen in Figure 3.3. The first step is query logging, which is responsible for publishing information about queries in execution to the index. This information is collected and stored in the index, and made available for the next step, History Analysis. During history analysis the raw information is post-processed to produce statistics of data usage in queries. The third step, Evaluation, determines the most profitable candidates from these statistics and suggests a site where this candidate can be created. The fourth step, Publish Candidate, publishes the candidate to the index as a false cache entry. This cache entry, if used by the optimizer during planning, will ultimately be turned into a real cache entry as a part of the fifth step, Cache Creation.

**Query Logging** Cache investment depends on a knowledge of what has been processed in the past. To do this we will gather information by logging queries. As a means of logging query and sub-query usage we will introduce query logging. Query logging will make use of the structure we call RSID to publish information to the index when a query is executed. Initially there are two approaches to this. We can either choose to send a message for every executing node in the algebra tree, as seen in Figure 3.5, or we can collect a batch of information which is then published to the index when the

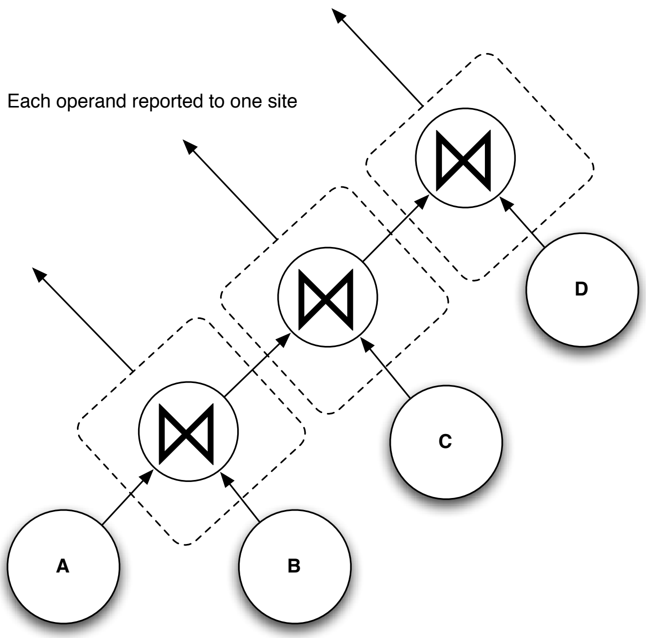


Figure 3.5: Query logging: Operands are reported individually to their own specific site.

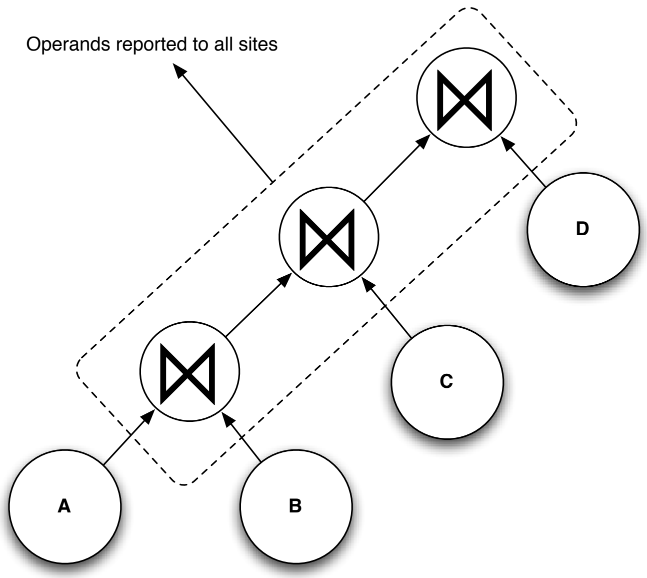


Figure 3.6: Query logging: Operands are reported as batch to all participating sites.

complete query has executed successfully. Sending a batch, as seen in Figure 3.6, will reduce message traffic but will require additional information about which sites had which algebra node assigned to them during execution. This information is implicitly given if every message is sent individually.

Before deciding upon how to log, we must also consider where to log. In order to adhere with the peer-to-peer design of DASCOSA, it seems natural to utilize the DHT for this logging. As explained in Section 2.7.1, the DHT is responsible for holding the distributed index. The distributed index can be seen as the system's collective consciousness. The index is primarily concerned with storing pointers to sites storing table fragments. The index does this by storing the pointers in the DHT with the table name as key. This means that for every table there is a site responsible; the table's indexing site. All index sites contribute to maintain the index as a whole. In contrast to the table lookups the index is created for, a query being reported usually contains more than one table. This gives us the choice of either choosing one of the tables with a hash function or logging to all of them. As a side note the latter has the advantage that more of the history may be preserved in case of site failure, but at the cost of a larger overhead. As an a priori decision we believe the cost of the extra messages could be just as harmful in terms of handling churn, but in the end we chose to not weight this as DASCOSA currently has no replication scheme for the tables either. It should be mentioned that too few messages is really not a problem. If a query logging message at some point is lost, the degradation of the history entry in question would be negligible. The history is after all only a rough estimate.

We now have two decisions to make, so we need to see if they are related.

One algebra node reports to one indexing site: This solution implies that for the join of tables A, B, C and D a hashing function has to select one the tables' indexing site. Such a function could be to always choose the site for the table name that comes first in alphabetical order. The query with tables A, B, C, and D would then be reported to the indexing site for table A, $Site_A$, and the query with tables B, C and D would be reported to the indexing site for table B, $Site_B$. In some swarms, $Site_A$ and $Site_B$ could in fact be the same site, but this is not guaranteed at all and can not be assumed. This case leads us to the problem with this solution. Individually the indexing sites may choose neither of the queries profitable for investment. In fact if they had had the complete picture, they would have suggested caching B, C and D. This would also be advantageous for the query A, B, C and D, and therefore profitable in sum.

One algebra node reports to all indexing sites: This combination solves the problem of the previous, in that one of the indexing sites would receive both queries and see the potential for caching B, C and D. The disadvantage with this solution lies in the overhead during execution, as this alternative would require reporting of the tuple counts of from every sub-operation to the final site.

All nodes report to all indexing sites: The advantage with this alternative is that just as the previous each indexing site is capable of doing the history analysis and it does not require any particular roles during execution as the previous. The huge disadvantage with this solution that gives it a worse overhead than the previous is that for a query with n relations it requires $\Omega(n^2)$ messages.

All nodes report to one indexing site: This alternative combines the advantages of the previous alternatives and this at a message count of only $\Omega(n)$ with n tables in a query. It does require all indexing sites to do history analysis, but none of them are doing any duplicate work so this could be considered an advantage in terms of load balancing as well. All indexing nodes have to be queried for table lookups anyways so there is no extra messages when doing lookups. This is the alternative that we consider the best for DASCOSA.

Now that we have decided on where the logging messages will sent from and to, we can decide on what they actually have to contain. The RSID of the data just produced is the first and most obvious component. However as we are sending one log message per operation, one RSID per message is sufficient. Non-PJS-operations are not logged as there is no point in doing cache investment for them as the optimizer is not capable of planning for their use. Secondly the cache investment needs to know where data is used. It is important to note that this is not necessarily the same site as the site reporting. We say that we have a data consumption-oriented history opposed to one that is production-oriented. For this reason we include which site the result was shipped to. The combination of RSID and site gives the picture of where data is used, but for the purpose of comparing cache candidates we also include the planned cost of the operation and size of the result produced. The contents of the log message are summarized in Table 3.4.

| **Field** | **Description** |
|---|---|
| RSID | The ResultSetIdentifier of the data produced |
| Site | The site the data is produced for |
| Cost | The planned cost of producing the data, as given by the optimizer. |
| Size | The actual size of the data produced |

Table 3.4: The contents of the log message.

**History Analysis** Every time an RSID is published to the index the value of every candidate in the index is updated and if not present the RSID is added to the candidate set itself. This means that for every site having reported the use of a table set, a weight is maintained. A candidates weight is incremented by the benefit has given to the queries logged. The benefit is defined in Equation 3.7.

$$Benefit = Max(PreviousCostAtSite - UseCost, 0) \tag{3.7}$$

A candidates weight represents the value of having the table set cached at that site. In addition the history also maintains the average reduction factor for the table set at every site. This allows for better estimation of the cost of using a cache entry, as seen in Equation 3.8. For instance if $Site_A$ repeatedly runs a query requiring all the tuples and $Site_B$ runs another query just as often, but the query from $Site_B$ asks the caching site to apply a constraint that only lets 10% of the tuples through then the cache should be stored closer to $Site_A$ than $SiteB$.

$$UseCost = TupleCount * AvgReduction \\ * Distance(Site, CandidateSite) \quad (3.8)$$

To maintain the query history we will use a technique called periodic aging by division[9]. An aging factor, $\alpha$, is a tuning parameter set between 0 and 1. Every time a query has been reported all the weights are multiplied by $\alpha$. A low $\alpha$ gives the most weight to recent queries. However, this also makes the system prone to transient changes. According to [9], the performance is better with high or low than with mean values of $\alpha$.

If an RSID reference ever falls under a given threshold, it is removed from the history. This is to ensure that only queries of a certain value will be kept in the history. This also ensures that the size of the history does not grow unbounded. If an RSID does not manage to stay alive, then its usage is deemed too low and therefore uninteresting to our analysis.

**Evaluation** The purpose of this step is twofold. First, the cache investment module must identify a good candidate for caching based on previous history analysis. The statistics from the history analyses can be interpreted in several ways. How much weight that should be given to previous usage or return on investment is determined by the policy used. Some proposed policies for identifying cache candidates are the reference counting policy and the profitable policy, which were described in Section 2.6.1 Second, a site must be identified for the cache to be created. This site must be located in such a way that the cache will be of most value to as many nearby sites as possible. This will be achieved through cluster analysis among known sites. Due to scalability reasons and being a P2P system, a site in DASCOSA does not necessarily know of every other site in the system, as mentioned in Section 2.7.1. The set of known sites to the index site is therefore the union of its neighbor set and the sites participating in the history.

Figure 3.7 shows sites located in the network coordinate system. While the coordinate system normally has more dimensions, we use two dimensions here for easier representation. The figure shows two floating sites A and D, and two clusters of sites B and C. If A finds a cache candidate and starts looking for a potential site to create this cache on it will detect the two

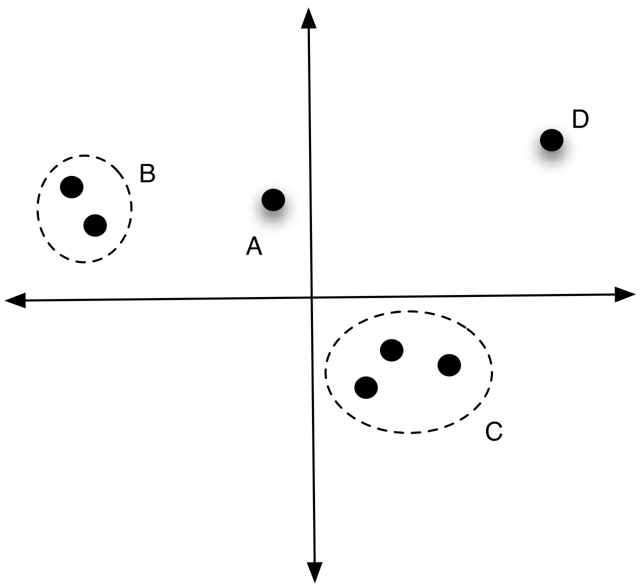


Figure 3.7: Clusters of sites.

clusters B and C. Of the two clusters, C would likely be chosen because it contains the most sites. Among the sites in the C, cluster a single site will be picked as the destination for the cache candidate. The cache investment module has no knowledge about the available space in the cache of sites. This might be a problem if the selected destination site has no free space left for the cache candidate. Perhaps the best solution here, is to leave the final choice for the cache replacement policy used by the cache manager.

**Publish Candidate** This is the step where the cache investment module will influence the query optimizer to possibly add the cache candidate during planning. The query optimizer stands free to determine if the cache candidate provided is profitable in the query execution plan. This hint is based on the RSID and site identified in the previous step, Evaluation. If a hint turns out to be not so good, the query optimizer will likely exclude it from the final query execution plan and no harm is done. On the other hand, should it decide to use the hint, this will lead to a cache miss during execution. Cache miss is a special case which then will need to be handled.

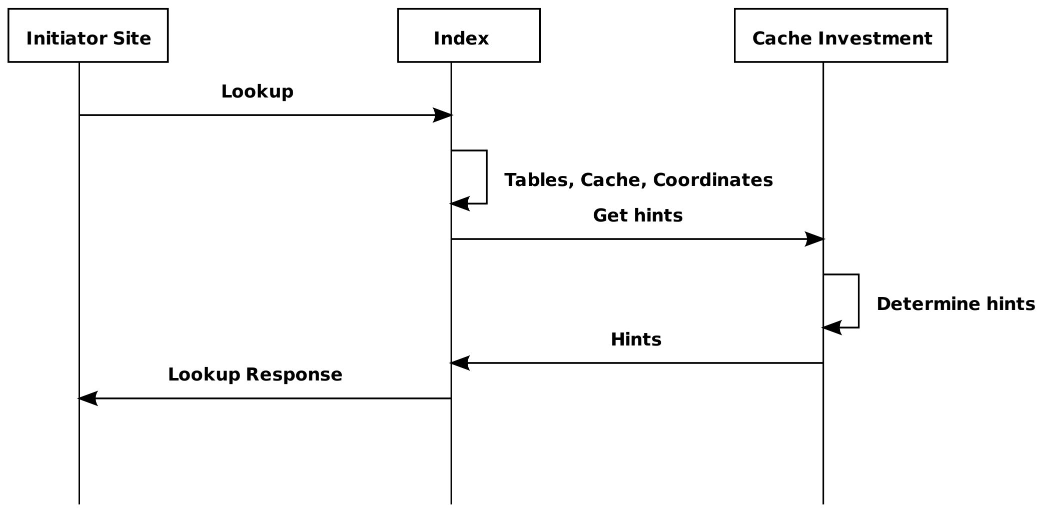


Figure 3.8: Sequence diagram for cache investment.

**Cache Creation** This final step handles the actual outcome. The query optimizer has found the cache candidate profitable enough to be a part of the final query execution plan. The caching candidate is now translated into a cache node in the algebra tree for the query and submitted along with an alternate plan for execution. If the executing site for the cache node does not have a cache entry corresponding to the RSID of the cache node it is called a cache miss. A cache miss is handled by executing the alternate plan provided. The alternate plan contains all operations required to construct the cache entry at the site that originally as supposed to contain the cached data. This finalizes the process of cache investment by creating a cache entry based on a hint provided by the cache investment module.

### 3.4.2 Implementation

An independent cache investment module was established on every site. This cache investment module was linked directly to the index manager on each site, so that it could be notified about arriving index lookups. The message flow can be seen in Figure 3.8. The main responsibility of this module is to process messages that report completed algebra nodes in the database. When an algebra node completes, a report is sent to the index site holding the first of the tables making up the result.

```
double adjustWeights(ResultSetIdentifier reportedRSI, NodeHandle
    reportedSite) {
  double totalBenefit = 0.0;
  for (NodeHandle site : resultSetUsage.get(reportedRSI).
      statistics.keySet()) {
    totalBenefit += benefit(reportedRSI, site, reportedSite);
    resultSetUsage.get(reportedRSI).statistics.get(site).weight
        += benefit(reportedRSI, reportedSite, site);
  }
  return totalBenefit;
}
```

Listing 3.1: Adjusting the weights of other entries and calculating their constribution to the reported query's weight.

Each processed report results in a weight increase for the query reported at the site it was reported for. This increase is determined by the value of having the RSID cached at that site. The increase is determined while adjusting the other entries' weights in the method defined in Listing 3.1. Adjusting the other entries is done by incrementing their weight with the benefit they would have given to the reported query had they existed as cache. Finally all entries in the history are aged. By aging we mean that that all entries will be given a weight reduction, resulting in a net decrease for the entries not related to the reported query. The benefit method referred to in Listing 3.1 is given in Listing 3.2. It is as close as possible a direct implementation of the function in equation 3.7.

```
double benefit(ResultSetIdentifier rsid, NodeHandle site,
   NodeHandle candidateSite) {
  Usage usage = resultSetUsage.get(rsid).statistics.get(site);
  double benefit = usage.cost - rsid.getTupleCount() * usage.
      getAverageReductionFactor() * linkStatistics.getLinkCost(
      candidateSite, site);
  if (benefit > 0)
    return benefit;
  else
    return 0.0;
}
```

Listing 3.2: The benefit of a cache candidate to a site.

When structuring history, a sub-query is represented as an RSID. This RSID points to an entry for each site where such a result has been produced. These RSID-site pairs are given a weight, which are used in the process to identify good caching candidates.

An implementation decision we needed to address was the question when the cache investment should do its work. There were three alternatives for how to do this.

1. Periodically.
2. On index lookup.
3. On query complete.

The first alternative, which is periodic cache investment processing seemed like a decent choice, but we were unsure about the overhead it would create and how to let it scale according to the current load. If we chose this approach, managing cache candidate hints would be an issue. This solution implied that the hints would have to be stored in a temporary location or added to the index.

The second alternative, on index lookup, fulfilled the role of providing hints on demand. This way we was sure that the hints given would be included in the planning process right away. As this was a part of an ongoing index lookup request, we could reply with a list of hints customized for the current query, reducing the overhead.

The third alternative, on query complete, would function much like the first alternative. While not as periodic, it would be called when a query was complete and thereby have no relation to the query in planning. The same management issues around cache candidate hint provision would exist.

Based on these considerations, we chose to use alternative two in our implementation. This solution was easy to integrate with existing framework in DASCOSA.

Once the decision on when to request and process hints was made, the next issue was to fabricate the reply. The reply would be one or more cache candidate hints with cost and site suggestions. Previous query results was ranked from most to least popular based on weight. These weights was then used to determine a weighted centroid for each cache candidate. The site suggestion was set to be the site closest to this centroid in our synthetic coordinate system. Three alternatives was considered for picking a threshold for caching candidates to be used. Any candidate that exceeded the threshold was to be a part of the reply to the query optimizer.

1. A factor of average cost of reported queries.

2. Top k candidates (globally or locally).

3. Knapsack algorithm.

The knapsack algorithm approach is obviously the one that utilize the cache the most, by tailoring the hint to maximize the profit. The problem is that keeping track of free size in caches on other sites is hard to do without breaking with the P2P ideology. This approach is the closest we get to Donald Kossmann's Reference Counting Policy[9].

In our solution we chose to use the top k candidates approach. As cache investment replied on each index lookup, the query optimizer was given k hints from each indexing site. Thus scaling the number of hints with the number of tables in the query. Giving away just the top ranking candidates for each request, provides the query optimizer with enough hints to work with. In our implementation, k was hard coded with the value of 2. A large k gives the optimizer more choices and as it focuses solely on the query in planning, too large a k would give little weight to the history.

During implementation we encountered a problem with how the alternate plan from the query optimizer. As described in Section 3.2.1, the alternate plan was generated with no cache included to avoid further cache miss. We thought this was a good decision at that time, because to implement this the query optimizer needed only select the next best plan. When finding an alternate plan for a cache hint we found that a previously generated plan was not sufficient, because the old plan could contain constraints or other elements not included in the hint. In the end we found it necessary to invoke the query optimizer a second time to generate the alternate plan. Since most information already had been collected during the early stages of the query optimizer, doing a second planning phase did not cost much.

## 3.5 Adaption for TPC-H Queries

TPC-H[38] queries need some adaption to be applicable for use with the query optimizer. Our optimizer does only fully optimize PJS-query. To optimize a full TPC-H query we must first identify the PJS-subtrees of a TPC-H query. This is done with a depth-first search. When optimizing subtrees this way, we will not know the target site for the sub-result because the complete plan has still not been decided. Our assumption is that an aggregate node, which is not a part of a PJS-query, often will be assigned to its child node. This

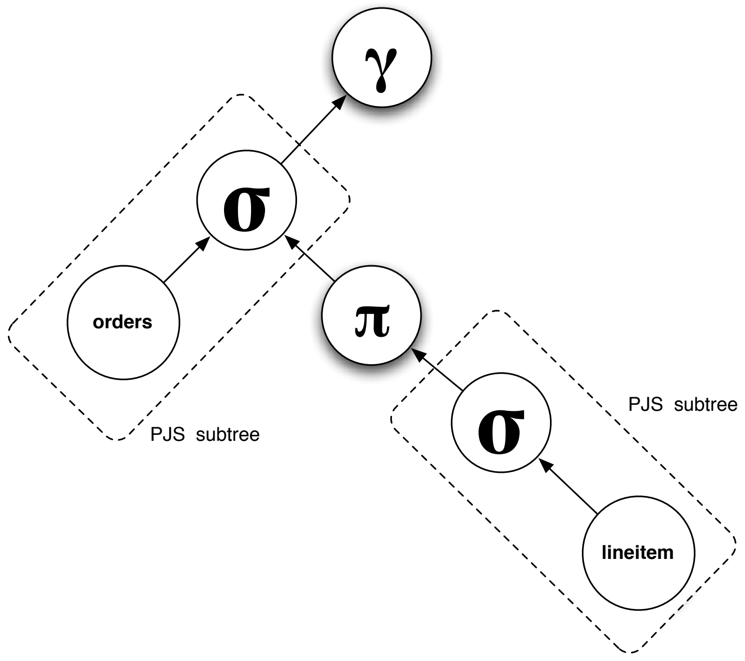


Figure 3.9: PJS subtrees in a TPC-H query.

aggregate node will in most cases reduce the result size enough that the cost of transporting the result is negligible. This is not a perfect solution, but it does make the optimizer able to handle more advanced queries than just those applicable for the PJS-restriction. Figure 3.9 shows the relational algebra tree for a TPC-H query. The query has nested constraints (in the project node) and aggregates. Because of this, the query is not a PJS-query.

# Chapter 4

# Evaluation

This chapter will give a thorough evaluation of our implementation. With this evaluation we seek to find out if cache investment is suited for semantic cache in a distributed database management system. We hope to see that cache investment pays off, by answering the following questions:

1. How does cache investment perform compared to explicit and implicit cache usage?

2. How is cache investment's performance affected by different distribution scenarios?

3. Is cache investment better suited to serve a system with multiple concurrently executing queries?

In order to answer these questions performance will be measured in execution time from a query is submitted to the result is produced and returned to the user. Cache hit as a measurement, while commonly used for data cache, will not be used. Our solution is designed for semantic cache, and as explained in Section 2.3.3, cache hit rate is not a good measure for this type of cache. We will see if geographic distribution affects the merit of caching and cache investment, by testing them in three network scenarios.

First we will describe the testing environment in which all our test cases will be done. We will explain the setup of machines with their respective specifications and simulated locations. We will describe the data set that is going to be used, and how this data set is distributed among the sites. Then we will explain each specific test case and how these will be evaluated according to our evaluation criteria.

## 4.1 Testing Environment

The test environment consists of 8 machines, either with setup A or B. Setup A has Intel(R) Core(TM)2 Duo processors with 2.33 GHz, 4.0 GBs of memory, and are running Ubuntu linux with kernel version 2.6.27. Setup B has Intel(R) Pentium(R) 4 processors with 3.0 GHz, 2.0 GBs of memory, and are running Ubuntu linux with kernel version 2.6.28. Sites 0-5 was running setup A, and site 6 was running setup B.

| **Table** | **Tuples** |
|---|---|
| Nation | 25 |
| Region | 5 |
| Supplier | 200 |
| PartSupp | 16000 |
| Customer | 3000 |
| Orders | 30000 |
| Part | 4000 |
| LineItem | 120515 |

Table 4.1: Tuplecount for each table.

To simulate geographic distance between the sites they are each routed through a server in another city with IPv6 tunnels. This provides us with a good approximation of the latency variance and throughput limitations that real distribution would have given. The tunnels are provided by a tunnel broker. Our tunnel broker is called Hurricane Electric[39] and provides tunnel locations from several cities in the United States, Europe and even Hong Kong. Figure 4.2 illustrates the tunnel locations. This technique of simulating wide spread sites worked well in the specialization project[10]. One just have to be aware that the tunnels also have a latency of their own, as the sites are not actually located in each city. This means that the round trip time between Amsterdam and Paris is actually the latency between Trondheim - Amsterdam - Paris - Trondheim and back again. For the purpose of our evaluation this is not a problem, but if one should compare our work with others this should not be forgotten. Another curiosity is that the sites without tunnels have a shorter latency to all sites, as they do not have to pay the initial cost of sending data to their tunnel endpoint. This results in Trondheim appearing as the center of the network.

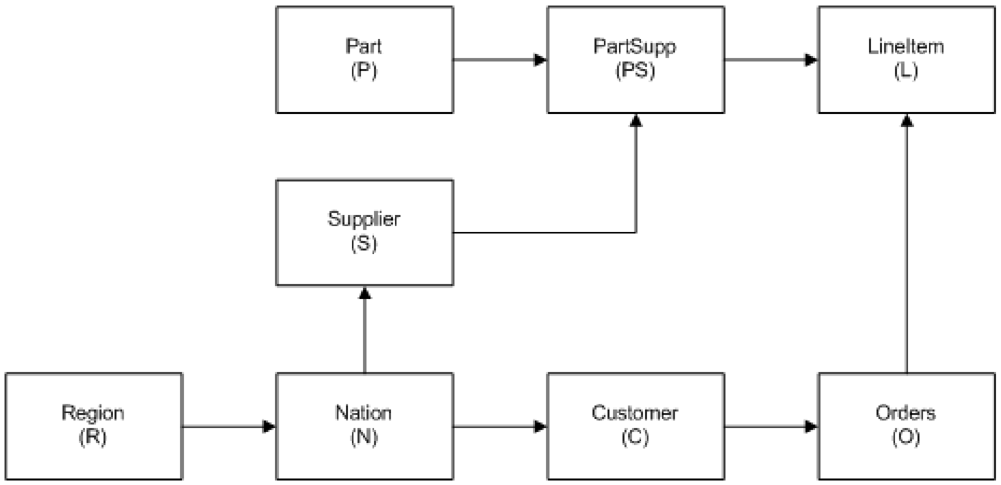


Figure 4.1: TPC-H database.

### 4.1.1 Data Set

To test the effect of cache investment on the performance of the semantic caching in DASCOSA we will use the data set provided by TPC-H Benchmark[38]. Figure 4.1 illustrates the distribution of tables within the data set. The benchmark is designed to be complex enough to simulate an industry workload for a decision support system. The data set has been generated with the 0.02 parameter, giving us a data set with scale factor (SF) 2% of the full size. The tables and an overview of how many tuples they each contain can be seen in Table 4.1. It would have been desirable with a larger scale factor, but unfortunately DASCOSA is an experimental system and it has some stability issues outside of our control. It was infeasible to complete all our tests within the desired time frame with a larger data set. We were forced to decide between cutting down on the test cases or the data set in order to finish on time. We prioritized the tests, because we did not know what results to expect for any of them or how they would relate. Still the 2% data set is large enough to allow the tuple counts of each table to be properly reflected in execution time of table scans. A simple two-computer LAN setup gave about 300ms for scanning customer, 2 000ms for scanning orders and 30 000ms for scanning the table lineitem.

Table 4.2 shows how the tables are distributed on the sites used in this evaluation. Empty sites can still be candidates for index maintenance and caching, even if they contain no table data when booted.

| Table | $Site_0$ | $Site_1$ | $Site_2$ | $Site_3$ | $Site_4$ | $Site_5$ | $Site_6$ |
|---|---|---|---|---|---|---|---|
| Nation | | | | | | 1 | |
| Region | 1 | | | | | | |
| Supplier | | | 1 | 2 | | | |
| PartSupp | 1 | | 3 | 4 | 5 | 2 | |
| Customer | 1 | | 3 | 4 | 5 | 2 | |
| Orders | 1 | | 3 | 4 | 5 | 2 | |
| Part | 1 | | 3 | 4 | 5 | 2 | |
| LineItem | 1 | | 3 | 4 | 5 | 2 | |

Table 4.2: Table fragments and site distribution.

### 4.1.2 TPC-H Queries

The queries in the TPC-H Benchmark set does not follow the PJS-restriction. We have dealt with this by optimizing only the parts of the queries that qualifies to the PJS-restrictions.

We will use the query set provided by the TPC-H Benchmark. Due to some limitations in the DASCOSA optimizer, not all tests applies. Queries that are not applicable are those where constraints are connected with boolean **OR**. TPC-H queries that qualify and will be used in this evaluation are shown in Table 4.3

A workload with a 80/20 distribution between queries will be used to simulate locality. That means that 20% of the queries will belong to a hot set and the remaining 80% is the cold set. At each execution there is 80% probability for the hot set and 20% for the cold set to be chosen. Within each set the probability is uniform. This distribution does a better simulation of the locality in a real workload than a completely uniform distribution. This is important as locality is the reason caching works in the first place. The different hot sets and corresponding cold sets are given in Table 4.3.

| # | **Hot-set** | **Cold-set** |
|---|---|---|
| 1 | 1, 6 | 2, 3, 10, 11, 12, 13, 14, 15, 16, 17 |
| 2 | 1, 13 | 2, 3, 6, 10, 11, 12, 14, 15, 16, 17 |
| 3 | 11, 12 | 1, 2, 3, 6, 10, 13, 14, 15, 16, 17 |

Table 4.3: Query workloads.

Most of the TPC-H queries define substitution parameters that are to be replaced with a random value from a given range. The benchmark specifies that these parameters are chosen with a uniform distribution[13]. This suits our implementation of cache investment very well as it always creates hints without any constraints. With highly selective queries and locality in the constraints we expect the non-investment alternatives to perform closer to cache investment. Of course a future implementation of cache investment should consider this as well.

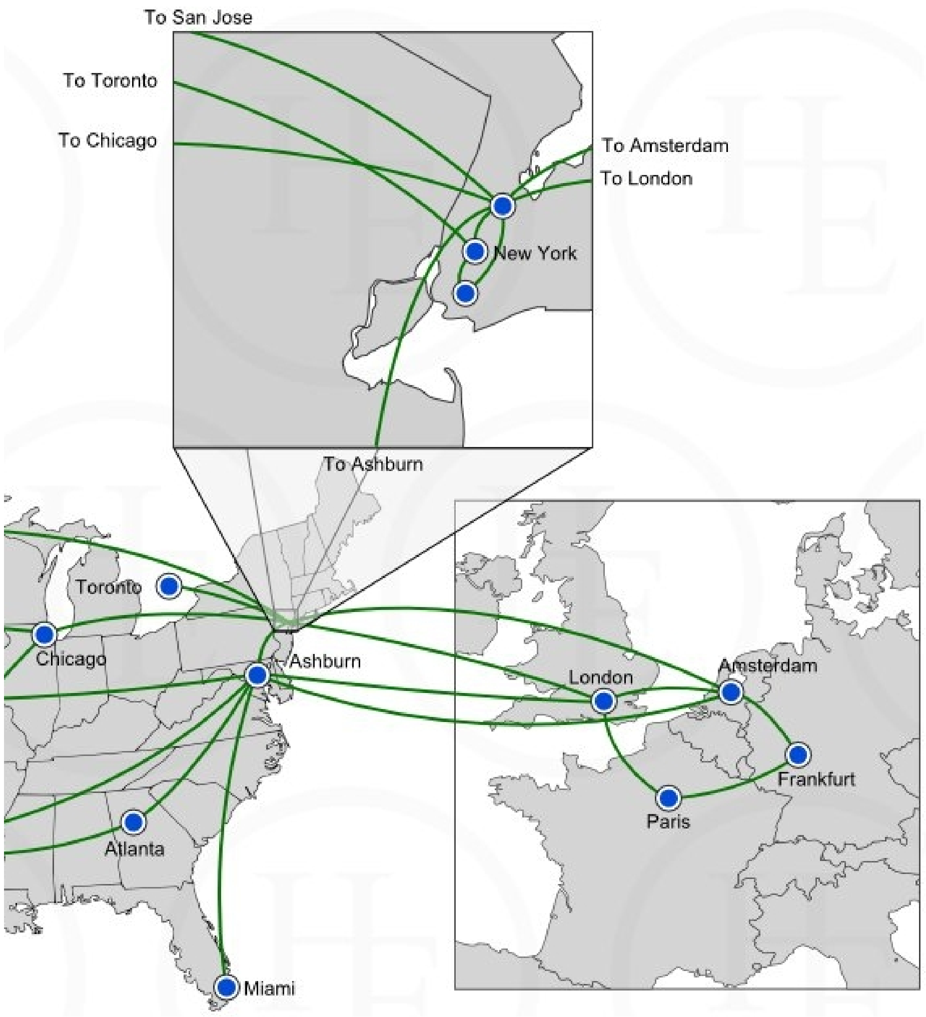


Figure 4.2: Network map[39].

## 4.2 Evaluation Criteria

The criterium we will use to evaluate our solution is execution time for queries. Execution time is elapsed time from the query is given to the result is produced. To show the effect of applying different techniques we will run test cases with the following parameters:

1. Implicit semantic cache.
2. Explicit semantic cache (our cache-aware optimizer extension)
3. Semantic cache investment

As explained in Section 2.5, implicit caching means that the optimizer does not consider cache when planning. During execution a node can check its site for a potential cache replacement. Cache hits occur because the same site is still considered the best for producing a result. This is how semantic caching is utilized in DASCOSA before we apply our solution. Explicit cache is our cache-ware optimizer extension. Here the perspective of cache entries are broadened by using the index to get complete overview. Cache entries are included as options during planning and if found profitable also a part of the final plan.

## 4.3 Test Cases

These four setups will be used for the sites. We aim to see how network distance and clustering affects our techniques for semantic cache. Network without tunnels will be referred to as local area network (LAN), and when tunnels are included this will be referred to as wide area network (WAN).

1. Heterogeneous distribution, WAN
2. Homogeneous distribution, WAN
3. Homogeneous distribution, LAN
4. Concurrency with heterogeneous distribution, WAN

For network distance between sites we will be using IPv6 tunneling to simulate distribution of the nodes. Clustering means that there can be more than one site at the same geographic location. Assigning cache candidates to a site within a cluster will be helpful for the whole cluster. No clustering means each the location of each site is unique within the system. When no network distance applies, the IPv6 tunneling will be turned off, and the sites will be using their native connections instead. In all four cases there will only be a single site executing queries. The last setup will be used to examine the effect of different origins of queries in the system, this forces the system to ship data in more than one direction.

| **Machine** | **Tunnel Location** | **RTT (ms)** |
|---|---|---|
| $Site_0$ | London | 103.00 |
| $Site_1$ | Trondheim | 0.27 |
| $Site_2$ | Amsterdam | 102.09 |
| $Site_3$ | Ashburn | 256.19 |
| $Site_4$ | Paris | 117.95 |
| $Site_5$ | Amsterdam | 104.53 |
| $Site_6$ | Trondheim | 0.17 |

Table 4.4: Machine setup for heterogeneous distribution.

**Heterogeneous Distribution, WAN** This test case will evaluate the general performance of our solution. The test environment will be set up with sites geographically distanced through IPv6 tunneling as seen in Table 4.4. The queries will be executed from a site in Trondheim, $Site_6$ which holds no data fragments.

**Homogeneous Distribution, WAN** This test case will evaluate the general performance of our solution when there is little variation in inter-site latencies, while keeping the geographically distance high. This makes all sites more or less equal, but still gives good cache entries the opportunity to have a great impact on performance.

**Homogeneous Distribution, LAN** One of the core ideas of the DASCOSA optimizer is to use knowledge about network distance to other sites to estimate the most effective transmission route for data. Cache investment is also dependent on this to determine good locations to position the cache. We will tie the sites in the system closer together to a single cluster located in a Trondheim, and see how this affects the execution time for queries.

**Concurrency and Distribution, WAN** The cache investment module is designed to detect patterns in previously executed queries. When there is little variations in the origin of queries this information is not very exciting because the flow of data will always point towards the one site executing queries. We will try to run queries on more than one site concurrently. More sites executing queries will distribute the demand for data more evenly, thereby creating good foundation for the cache investment to determine centroids for cache candidate locations. We hope to see that cache investment can identify cache candidate sites and place this on a location such that more than one site can make use of it and increase performance compared to the previous case with heterogeneous distribution.

## 4.4 Results

This section will show the results of the test cases explained above. The results will be analyzed and reasons will be given for the observed behavior. The results was produced using 3 workloads with 5 repetitions for each. The workloads can be seen in Table 4.3. The total amount of queries was 3000 per test case.

### 4.4.1 Heterogeneous Distribution, WAN

The results of this test are shown in Figure 4.3. The execution times for each workload with standard deviation given are shown in Figure 4.4. As can be seen in the graph, there is little difference between implicit and explicit use of cache. By studying the details of the test run we can see that the coordinates for the sites remained stable and unchanged throughout the test, which lead to little variation in site selection and high determinism. Explicit use of cache chose to use the same sites as the implicit cache run, and thereby used the same cache entries for most of the tests. The small advantage explicit cache has over implicit is based in those cases where another site not normally detected by the planner would contain a favorable cache. Explicit cache

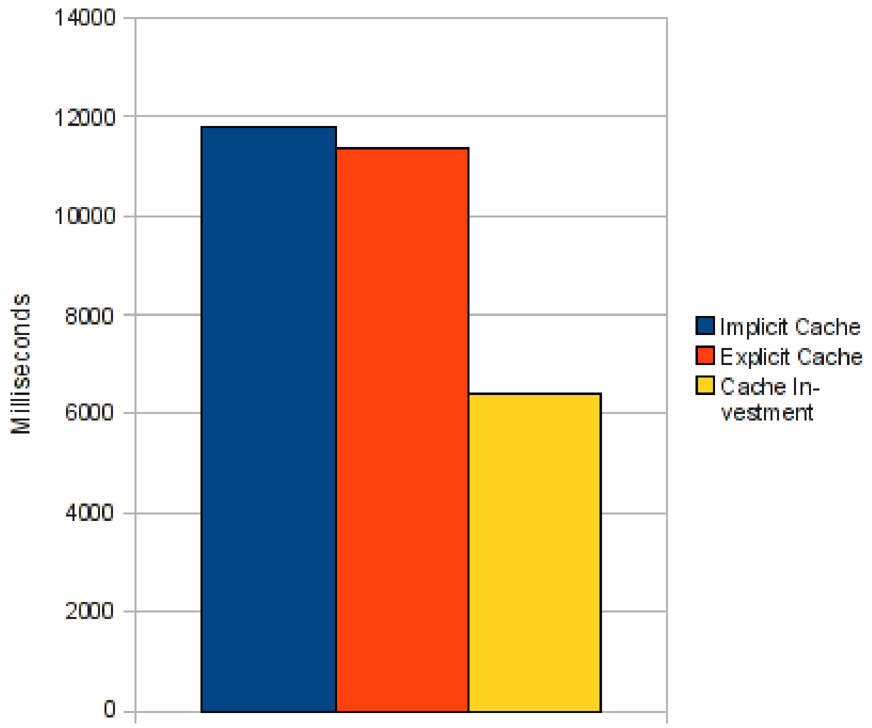
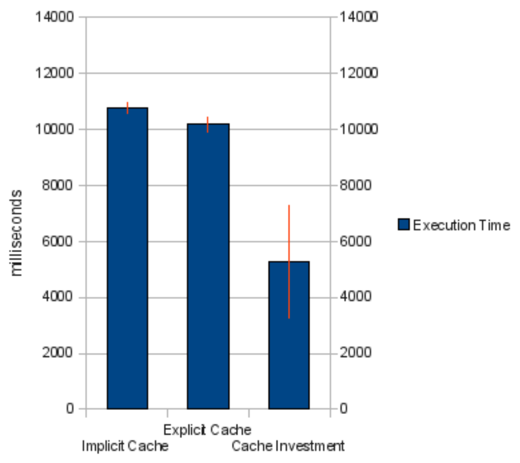


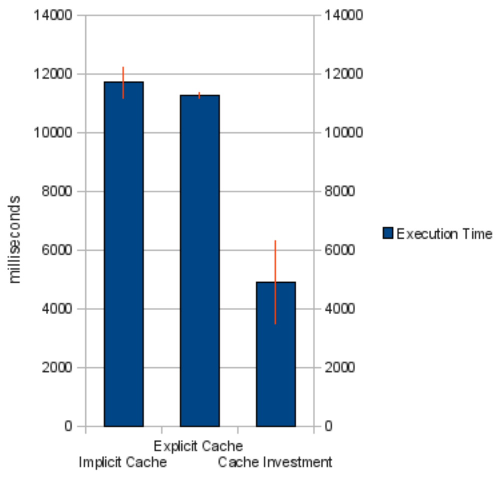
Figure 4.3: Execution time for setup with hetereogeneous distribution on WAN.

would be able to detect this and exploit the cache entry created by another query, and implicit cache would not.

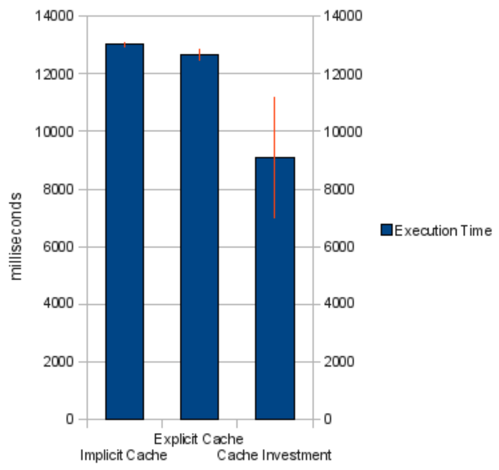
The leap from those two techniques to cache investment requires some explanation. Initially it will operate like the other two techniques, but after a few runs when the history has been allowed to grow large enough for patterns to be detected this changes. Cache investment will calculate the centroid for the candidates it is going to suggest. Based on the origin of queries it will detect a clustering of sites around this location. Therefore it will generate a hint that will satisfy as many of the queries in the history as possible and place this somewhere suitable in the cluster. The site executing queries will receive a very good cache right at its doorstep and execution time for queries thereafter will experience a drastic experience gain. These results are very dependent on their environment, and we expect to see different results if there are no clustering among the sites, or if all sites are put in the same



(a) Workload 1



(b) Workload 2



(c) Workload 3

Figure 4.4: Standard deviation for each workload for heterogeneous WAN.

cluster with little or no network distance.

### 4.4.2 Homogeneous Distribution, WAN

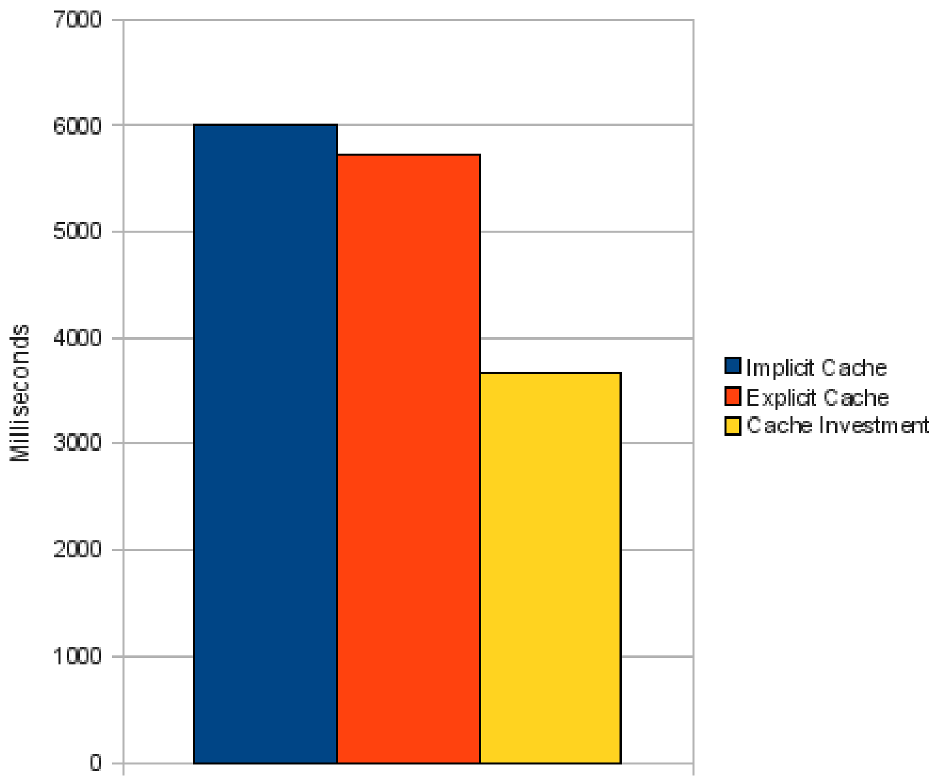
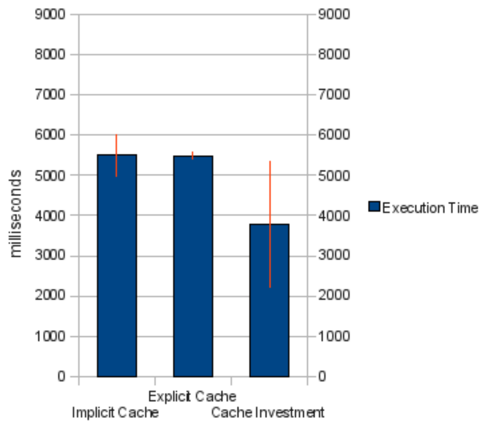
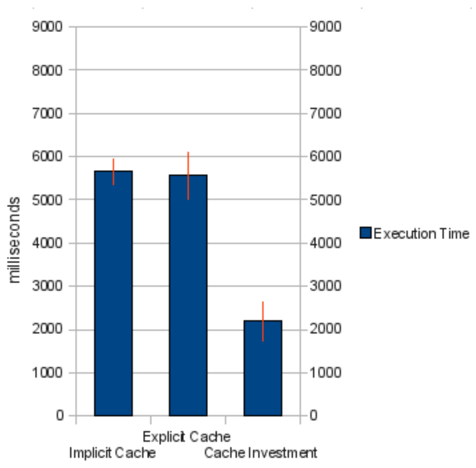


Figure 4.5: Execution time for setup with homogenous distribution on WAN.

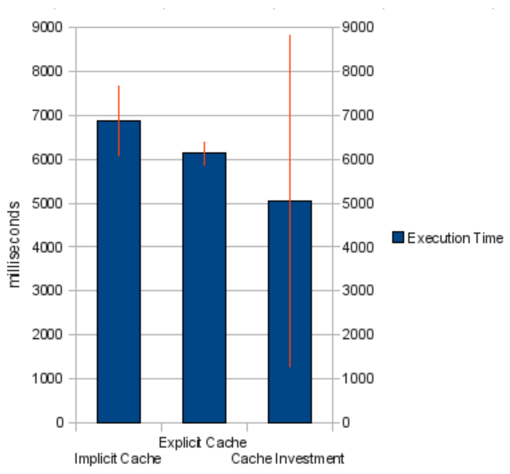
In this case all the sites were setup up to route the traffic through Amsterdam. This gave the data a much larger distance to travel between sites, and gave communication cost a much bigger role to play during execution. The results of this test can be seen in Figure 4.5. The execution times for each workload with standard deviation given are shown in Figure 4.6. As given in Figure 4.6c, the standard deviation for cache investment on workload 3 is very large. To explain this we analyzed the logs and made graphs to uncover a pattern. What we found was that one particular batch of queries for workload 3 showed consistently longer execution times for similar plans. Because



(a) Workload 1



(b) Workload 2



(c) Workload 3

Figure 4.6: Standard deviation for each workload for homogeneous WAN.

of this we reason that this was one unlucky batch stricken by an incident at the tunnel broker or in our local network that drastically increased the execution time required to finish. With this batch removed, the test shows normal behavior. This is shown in Figure 4.7.

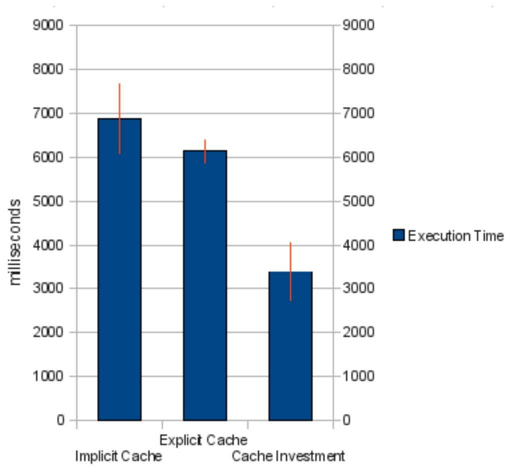


Figure 4.7: Standard deviation for workload 3 for homogeneous WAN with bad query batch removed.

In general explicit cache runs better than implicit cache, but is unable to ”improvise” during cache creation like cache investment. Cache investment makes use of its cache hinting feature to create cache entries for operations not previously considered on this particular site. Common to both implicit and explicit cache is that they consider only previously produced results for caching. Cache investment is also capable of suggesting a cache entry that is more general and therefore more applicable for future queries. The combination of more general cache entries and cache located closer to where it will be used is the reason for the speedup in cache investment.

### 4.4.3 Homogeneous Distribution, LAN

In this case the tunnels was switched off. The sites was located on the same local area network. The results of this test are shown in Figure 4.8. The results lived up to our expectations. When there is equal delay between all nodes there is little to gain from smart site selection. Caching will have a small impact if local cache is used often or if the operation cost of producing

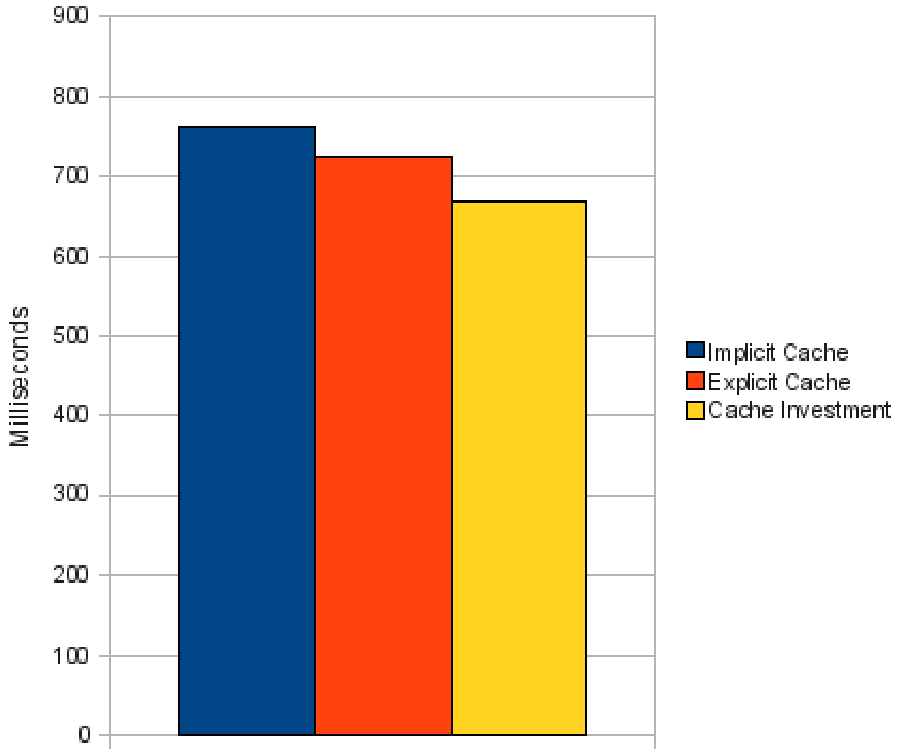
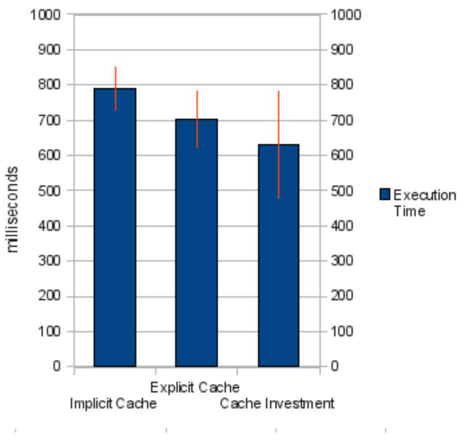


Figure 4.8: Execution time for setup with homogeneous distribution on LAN.

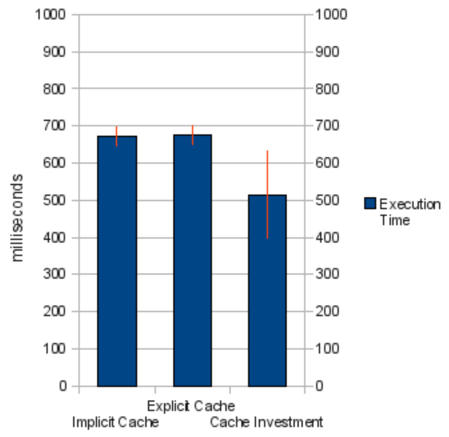
a result is expensive. From the graph we can read that explicit caching has a marginal advantage over implicit caching and cache investment takes this one small step further. While cache investment does not hurt performance in this setting, there is also little to gain. In a local network, communication cost is just one of many factors. Cache investment seeks to place cache so that it is near as many sites as possible, this criteria is not applicable in a local area network the distance is the same for all sites.

The execution times for each workload with standard deviation given are shown in Figure 4.9. Compared to the execution times the standard devation is not that bad, but compared to the difference between each technique it is apparent that more tests are required to confirm the trend in Figure 4.8.

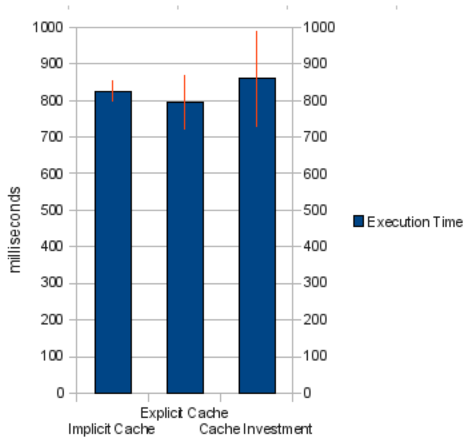
This case confirms our theory that cache investment will have greater impact in a geographically distributed setting. It also suggests that more tests needs to be done in order to reveal the cases where cache investment



(a) Workload 1



(b) Workload 2



(c) Workload 3

Figure 4.9: Standard deviation for each workload for homogeneous LAN.

may degrade performance.

### 4.4.4 Concurrency with Distribution, WAN

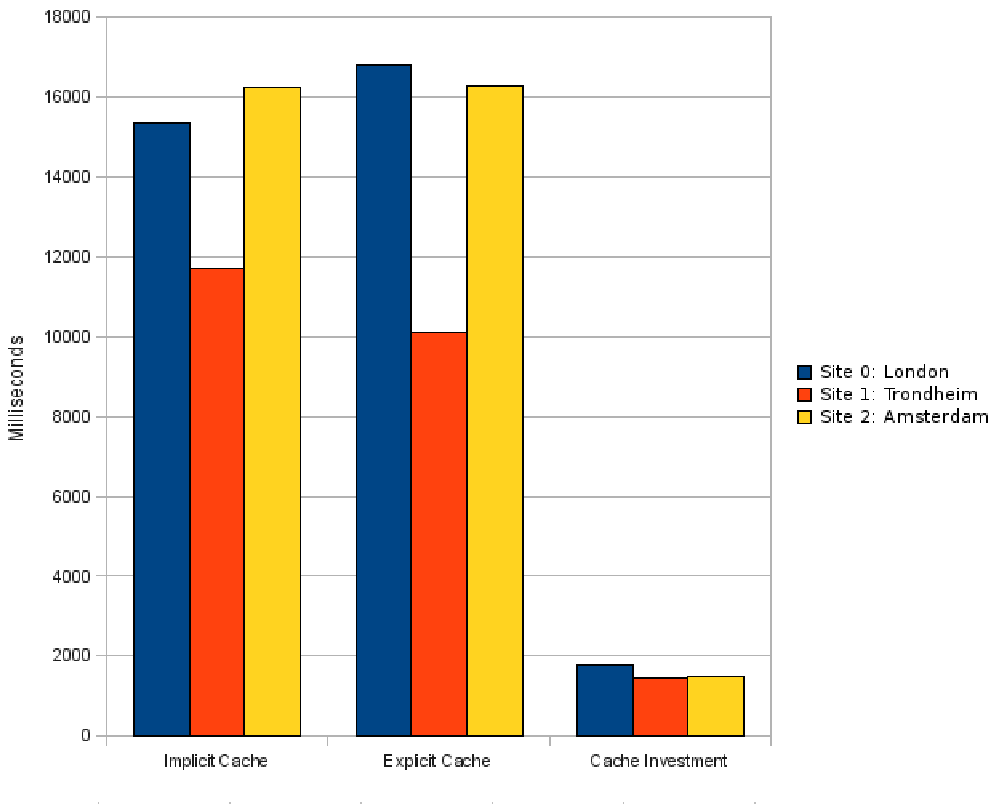
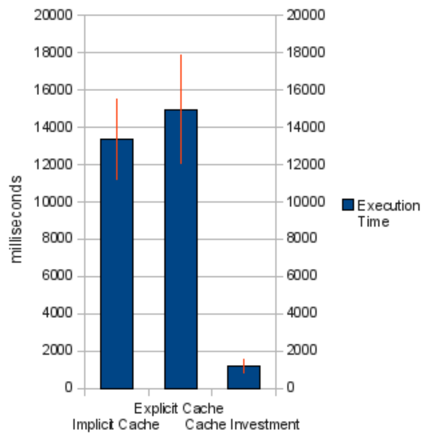
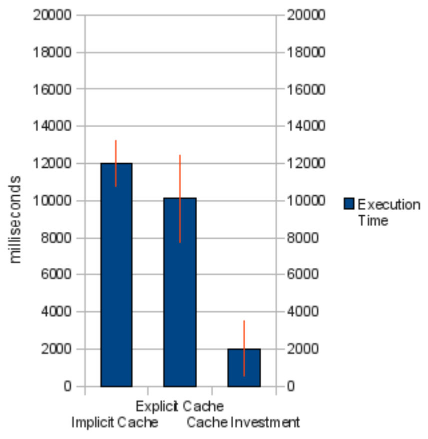


Figure 4.10: Execution time for setup with concurrent sites on WAN.

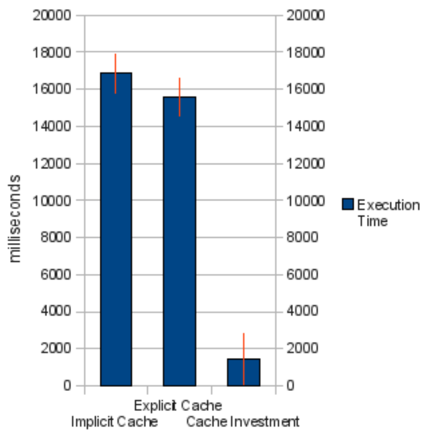
This test attempted to measure the effect of concurrency and cache investment. By concurrency, we mean that more than one site in the system is executing queries at the same time. When queries are issued from different locations like this, it puts more pressure on the system's ability to adapt its execution plans to the changing demand for data. No site use the same workload at the same run. As usual the cache is reset between each run. Worth noting is that both workload 1 and 2 have TPC-H query 1 in their hot-set.



(a) Site 0 Workload 1



(b) Site 1 Workload 2



(c) Site 2 Workload 3

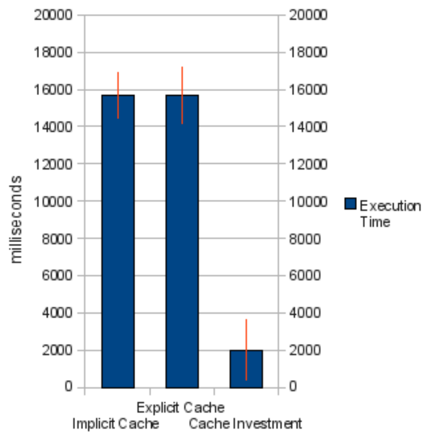
Figure 4.11: Standard deviation for concurrency test case 1.

Figure 4.10 shows the results of the tests. This is the case where cache investment excels like no other. The execution times with standard deviation given are shown in Figure 4.11, Figure 4.12, and Figure 4.13. The standard deviation for this test is fairly within limits to conclude that cache investment handles concurrency better than explicit and implicit caching.

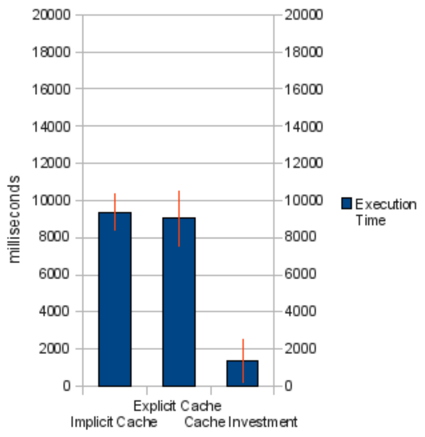
As the astute reader may have noticed, the test case for heterogeneous WAN and this one use the same location setup for their sites. Therefore, we would have expected the execution times for each caching technique to be longer because of the higher system load. This is in fact not the case for cache investment. We suspect two possible reasons for this behavior. One, cache investment is capable of doing bigger investments when more sites are demanding data. Two, the test cases are in fact not comparable, as they were executed at several days apart and the situation at our tunnel broker service might have changed.

### 4.4.5 Comparison

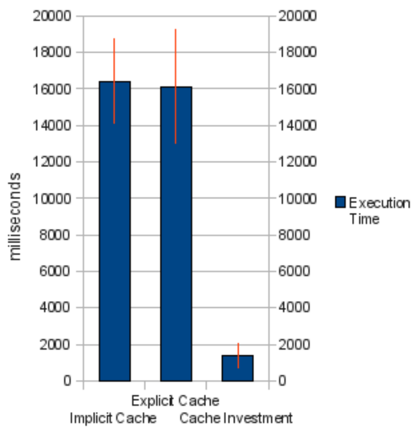
Table 4.5 shows the average speedup of the techniques compared to implicit caching. Latency is average latency in setup. As can be seen in the results, the speedup increases with the average latency value. This can be explained by the fact that higher latency leaves more to be gained from good cache



(a) Site 0 Workload 2

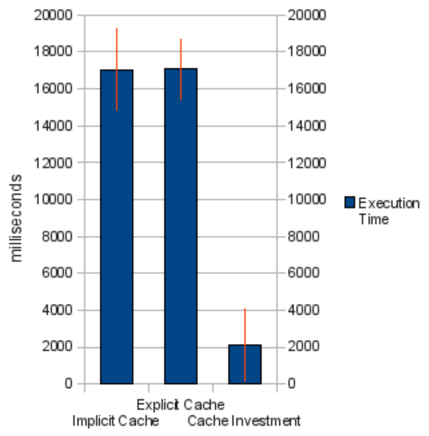


(b) Site 1 Workload 3

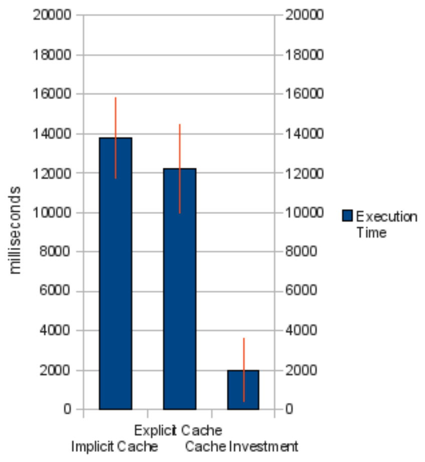


(c) Site 2 Workload 1

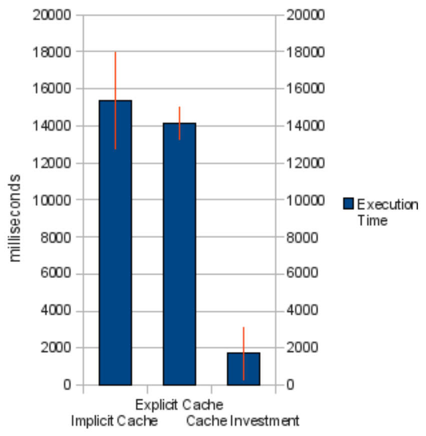
Figure 4.12: Standard deviation for concurrency test case 2.



(a) Site 0 Workload 3



(b) Site 1 Workload 1



(c) Site 2 Workload 2

Figure 4.13: Standard deviation for concurrency test case 3.

placement. Concurrency adds to that even further because in that case more than one site benefit from good placement of cache.

| **Test Case** | **Latency (ms)** | **Expl. Cache** | **Cache Inv.** |
|---|---|---|---|
| Heterogeneous distribution, LAN | 0.22 | 5% | 12% |
| Homogeneous distribution, WAN | 99.89 | 5% | 39% |
| Hetrogeneous distribution, WAN | 114.01 | 4% | 46% |
| Concurrency with hetero. dist., WAN | 114.01 | 3% | 89% |

Table 4.5: Speedup compared to Implicit Cache.

# Chapter 5

# Conclusion

Distributed databases is an important field in database research and development. The field has experienced an increase of interest in the recent years, mainly due to new demands on the database to handle larger data volumes and more users. This sets new requirements on efficient query processing, an important challenge which must be properly addressed before distributed databases can become efficient. DASCOSA[40] is a DBMS based on the peer-to-peer paradigm, designed to facilitate research into distributed databases. It is being developed at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

The effectiveness of query processing is a major component of a DMBS's performance. When we began our specialization project, DASCOSA had no dedicated query optimizer. During the project period we developed a distributed query optimizer based on the R*-algorithm.

In this project we have addressed semantic caching in distributed databases. We have designed an advanced caching strategy and implemented it for the DASCOSA DBMS. Semantic cache investment is an active caching strategy that deduces good caching candidates from query history and suggests these to the query optimizer. Our implementation was done in three phases, each phase designed to extend the capabilities of DASCOSA towards the goal of semantic cache investment. These three phases added selection support, query optimizer cache support and finally cache investment functionality.

We have shown that the concept for cache investment can be successfully deployed in a distributed database with peer-to-peer architecture and semantic cache. We have also shown that cache investment integrates well with an existing query optimizer solution such as the query optimizer in DASCOSA. After doing the switch from data cache to semantic cache it is still possible to do cache investment when proper query matching is applied.

Evaluation has shown improvement in performance when using semantic cache investment. The degree of performance boost from using semantic cache investment depends on the environment in which the distributed database exists. We have explored how network topology and concurrency affects caching in distributed databases. Results have shown that cache investment is effective. The effect on performance varies with changes in topology and concurrency, but shows an overall speedup in all cases. Cache investment is most effective in Wide-Area-Network systems with concurrent query execution. Our tests have also showed that the improvement of cache investment increases with both more latency and concurrency independently. This shows that cache investment takes into account locality between both sites and queries.

This solution has only explored the effects of cache investment on read performance. If the database is to do both reads and updates, then the issue of cache invalidation is introduced. DASCOSA's cache module does not currently handle updates, so neither does the cache investment. How cache investment behaves in such a setting we leave as work that will need to be addressed in the future. Still our results are interesting as data processing in data warehouses and other applications with very large data volumes is often done in phases. Numerous read only operations, like the one in our case, can be processed in a row before the system moves on to updates.

# Further Work

DASCOSA does now have a query optimizer and caching capability built around the concept of cache investment. There are many ways to take this work further.

One thing that struck us during our work on cache investment was the limitations of the cost model. The cost model does not perform good cost estimates for operations. This affects both the generation of the query execution plan and the cache replacement algorithm.

The query optimizer and cache in DASCOSA does not support all algebra operators. This has been cut due to our project scope and time frame. Extra operator support could very well lead to better query execution plans and better cache utilization.

Query matching is another field with possibilities for enhancement. Our solution does not support remainder queries. Remainder queries together with more precise query containment would be the next step in maturing caching in DASCOSA. This would require a better cost model.

There is room for improvements in cache investment in regard to cache candidate identification. Our solution is using a weighted sum function to identify centroids, but this does not consider size of the cache candidate nor the space in the cache for the respective sites. Currently the cache investment does not consider locality in selections. For future work we believe it to be worthwhile investigating the possibilities of doing this with a modified R-tree index with one dimension for each column of every table.

The cache investment module currently does not do load balancing. In a future version, sites publishing cache entries should include the size of their cache, how full it is and the value of of other cache entries.

# Bibliography

[1] M. Tamer Öszu and Patrick Valduriez. *Principles of Distributed Database Systems* (Prentice Hall), 1999.

[2] Yannis E. Ioannidis. Query optimization, 1996.

[3] Surajit Chaudhuri. An overview of query optimization in relational systems. In *In PODS*, pages 34–43. 1998.

[4] Angela Bonifati, Panos K. Chrysanthis, Aris M. Ouksel, and Kai-Uwe Sattler. Distributed databases and peer-to-peer databases: past and present. *SIGMOD Rec.*, 37(1):5–11, 2008. ISSN 0163-5808. doi:http://doi.acm.org/10.1145/1374780.1374781.

[5] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems: Fourth Edition* (The MIT Press), 2005. ISBN 0262693143.

[6] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995. ISSN 0163-5964. doi:http://doi.acm.org/10.1145/216585.216588. URL `http://portal.acm.org/ft_gateway.cfm?id=216588&type=pdf&coll=GUIDE&dl=GUIDE&CFID=64279817&CFTOKEN=61037937`.

[7] Philip Machanick. Approaches to Addressing the Memory Wall. Technical report, School of IT and Electrical Engineering, University of Queensland, 2002. URL `http://www.itee.uq.edu.au/~philip/Publications/Techreports/2002/Reports/memory-wall-survey.pdf`.

[8] Shaul Dar, Michael J. Franklin, Björn THór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of VLDB'1996*. 1996.

[9] Donald Kossmann, Michael J. Franklin, Gerhard Drasch, and Wig Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000. URL `http://citeseer.ist.psu.edu/kossmann00cache.html`.

[10] Konrad Giæver Beiske and Jan Bjørndalen. *DASCOSA Query Optimizer*. preproject, Norwegian University of Science and Technology, Dec. 2008.

[11] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger, and Paul F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, pages 31–47 (Springer), 1985.

[12] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. *SIGCOMM Comput. Commun. Rev.*, 34(1):113–118, 2004. ISSN 0146-4833. doi:http://doi.acm.org/10.1145/972374.972394.

[13] Transaction Processing Performance Council (TPC), Presidio of San Francisco Building 572B Ruger St. (surface) P.O. Box 29920 (mail) San Francisco, CA 94129-0920. *TPC Benchmark H (Decision Support) Standard Specification*, 2.8.0 edition, 2009. URL `http://tpc.org/tpch/spec/tpch2.8.0.pdf`.

[14] Chungmin M. Chen and Nick Roussopoulos. The query optimizer of adms. In *Fourth Intern. Conference on Extending Database Technology, Cambridge, UK, March 28-31, 1994*. 1994.

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

[16] Parke Godfrey and Jarek Gryz. Answering queries by semantic caches. In *DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 485–498 (Springer-Verlag, London, UK), 1999. ISBN 3-540-66448-3.

[17] Michael Stonebraker and Joseph M. Hellerstein. What goes around comes around. In *Readings in Database Systems*, chapter 1 (The MIT Press), 4 edition, 2005.

[18] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/356887.356892.

[19] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1070838.1070856.

[20] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal: Very Large Data Bases*, 5(1):35–47, 1996.

[21] Björn Þór Jónsson, María Arinbjarnar, Bjarnsteinn Þórsson, Michael J. Franklin, and Divesh Srivastava. Performance and overhead of semantic cache management. *ACM Trans. Internet Technol.*, 6(3):302–331, 2006. ISSN 1533-5399. doi:http://doi.acm.org/10.1145/1151087.1151091.

[22] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA), 1996. ISBN 1-55860-382-4.

[23] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000. ISSN 0360-0300. doi:http://doi.acm.org/10.1145/371578.371598.

[24] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43 (ACM, New York, NY, USA), 1998. ISBN 0-89791-996-3. doi:http://doi.acm.org/10.1145/275487.275492.

[25] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems* (McGraw-Hill), 2003.

[26] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computer Surveys*, 1993.

[27] Peter Pietzuch, Jonathan Ledlie, and Margo Seltzer. Supporting network coordinates on planetlab. In *WORLDS'05: Proceedings of the 2nd*

*conference on Real, Large Distributed Systems*, pages 19–24 (USENIX Association, Berkeley, CA, USA), 2005.

[28] Chungmin Melvin Chen and Nicholas Roussopoulos. The implementation and performance evaluation of the adms query optimizer: integrating query result caching and matching. In *EDBT '94: Proceedings of the 4th international conference on extending database technology*, pages 323–336 (Springer-Verlag New York, Inc., New York, NY, USA), 1994. ISBN 3-540-57818-8.

[29] Sheldon Finkelstein. Common expression analysis in database applications. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 235–245 (ACM, New York, NY, USA), 1982. ISBN 0-89791-073-7. doi: http://doi.acm.org/10.1145/582353.582400.

[30] Xiaolei Qian. Query folding. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 48–55 (IEEE Computer Society, Washington, DC, USA), 1996. ISBN 0-8186-7240-4.

[31] FreePastry, `http://freepastry.org/`, 2007.

[32] K. Nørvåg, E. Eide, and O.H. Standal. Query planning in P2P database systems. *Digital Information Management, 2007. ICDIM '07. 2nd International Conference on*, 1:376–381, Oct. 2007. doi:10.1109/ICDIM.2007.4444252.

[33] Apache Derby, `http://db.apache.org/derby/`, 2007.

[34] Michael Stonebraker et al. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[35] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981. ISSN 0362-5915. doi:http://doi.acm.org/10.1145/319628.319650.

[36] Norvald H. Ryeng, Jon Olav Hauglid, and Kjetil Nørvåg. Distributed semantic caching. Technical report, IDI, 2008.

[37] Jon Olav Hauglid, Kjetil Nørvåg, and Norvald H. Ryeng. Efficient and robust database support for data-intensive applications in dynamic environments. In *Proceedings of ICDE*. 2009.

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

[39] Hurricane Electric, `http://tunnelbroker.net/`, 2008.

[40] DASCOSA Project, `http://research.idi.ntnu.no/dascosa/`.