NTNU
Innovation and Creativity

PORDaS
Peer-to-peer Object Relational Database System

Eirik Eide
Odin Hole Standal

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Kjetil Nørvåg, IDI

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

Problem Description

The master thesis is about the Peer-to-peer Object Relational Database System (PORDaS), which is a distributed database system where the connected nodes are organized in a distributed hash table (DHT). The purpose of the thesis is to study query processing in a P2P environment, and to look at methods for increasing the performance. This means examining parallel execution and the use of replication.

Assignment given: 20. January 2006
Supervisor: Kjetil Nørvåg, IDI

Abstract

This master thesis presents PORDaS, the Peer-to-peer Object Relational Database System. It is a continuation of work done in a project of fall 2005, where the foundation for the thesis was laid down. The focus of the work is on distributed query processing between autonomous databases in a structured peer-to-peer network.

A great deal of effort has gone into compiling the theoretical foundation for the project, which served as a basis for assessing alternative approaches to introducing a query processor in a peer-to-peer database.

The old PORDaS version was extended to include a simplified, pipelined query processor capable of joining tables. The query processor had two different execution strategies, the first was performing join operators at the requesting node and the second was performing join operators parallel among the nodes participating in the query. Experiments which ran PORDaS on a cluster of 36 computers showed that there are room for improvements even though the system was able to perform all the tests.

Keywords : peer-to-peer, distributed hash tables, distributed databases, distributed query processing

Contents

List of Figures

Chapter 1

Introduction

Structured overlay networks have emerged as a promising infrastructure for building scalable and robust distributed systems. This thesis resumes the work of PORDaS, a project of fall 2005, which laid the foundation for a distributed database system in a structured peer-to-peer environment. The focus is on interconnecting independent, autonomous database systems to form a larger database system with extensive query processing capabilities while keeping data in their natural habitats.

The first part of the work will be to study relevant literature in the field of distributed query processing. Following this study, various alternatives for a distributed query processor will be compiled and evaluated. Some of the conclusions from the evaluation will be tested by adding a distributed query processor to the previous PORDaS version. The new features will be tested to see if the goals are met.

The background chapter introduces the basic theory on which the master thesis is based. It contains sections on distributed databases, object relational databases, peer-to-peer systems, distributed hash tables and a synopsis of previous and related work. The analysis chapter goes in further detail on distributed query processing. The PORDaS chapter presents the final PORDaS version, which is followed by a chapter on testing and results that gives a summary of the data gathered through various simulations. The final chapter concludes the master thesis and gives an outline for future work.

There are a few people who deserve credit for their contributions to the PORDaS project. First of all, Kjetil Nørvåg, who is the initiator and guide of this master thesis, has given valuable insights and direction on the theory and goals of the project. Svein Erik Bratsberg supplied helpful advice on distributed query processing. Kai Torgeir Dragland helped on accessing the Endless cluster, and Erlend Hammer gave access to the computer lab used for testing.

Chapter 2

Background

The purpose of this chapter is to give a brief presentation of subjects of relevance to the master thesis. There will be a presentation of distributed databases, object oriented databases and peer-to-peer systems. The previous work done on PORDaS in the fall project of 2005 is given as a short summary in order to keep this report self contained. The chapter ends with a section about related work.

2.1 Distributed Databases

It is assumed that the reader is familiar with the basic concepts of databases and distributed systems, as these will not be treated here. The interested reader is referred to [1] for a survey of databases. PORDaS will be built on many of the ideas provided in the field of distributed databases. It will share some of their properties, such as network transparency and having a query processor that copes with data scattered in many locations.

A distributed database [2, 3] is a set of logically interrelated databases spread in a physical network. The data stored in the system may be related even if it resides on different sites. There are various approaches to implementing a distributed database, though their preferable properties can be defined as a set of transparencies. A transparency in a system is a construct that shields its implementation details by providing a semantically higher interface. These are location transparency, performance transparency, replication transparency, transaction transparency, fragment transparency, schema change transparency and local DBMS transparency. Location transparency makes it possible to ask for data without knowing its location. Performance transparency refers to the idea that the performance of a query should be independent of which site the query was sent from. Replication transparency means that if there are replicas in the system, answers will not contain duplicate entries. Additionally, an answer will be produced as long as a replica remains alive. Transaction transparency implies that a transaction either commits or aborts, even though a query might involve more than one site. Fragment transparency makes it possible to divide a relation into fragments and spread these fragments to different sites. Schema change transparency implies that when data is inserted or removed from the system, the update is reflected at all the sites. Local DBMS transparency guarantees the operation of the distributed database despite the existence of any local DBMS working on the data at a site.

An obvious advantage of distributed databases is the ability to gather the resources spread across multiple sites in a single interface. This is especially useful when interconnecting legacy systems. Distributed databases have the promise of improved performance, which is due to two reasons. The first is data localization, which means that data can be fragmented or replicated close to sites where it is needed. The second is the use of parallelism, where queries can be split into smaller parts which can be resolved simultaneously. Multiple queries can also be handled at the same time.

Another advantage is the possibility of higher reliability through the use of replicas. Expanding the system is also easier than in the centralized case; adding a new site is usually a matter of registering it. However, there are limits to how far a distributed database can be scaled without penalties to performance due to increased overhead.

Distributed Query Processing

A distributed query processor is the component that resolves queries. The resolution process differs somewhat from the ones used in centralized databases. This is because there are more parameters to consider, like dealing with resource location, network delay, fragmentation of tables across sites and the existence of replicas. The steps typically involved in a distributed query processor are illustrated in figure 2.1. The first three are performed locally, while the last is distributed.

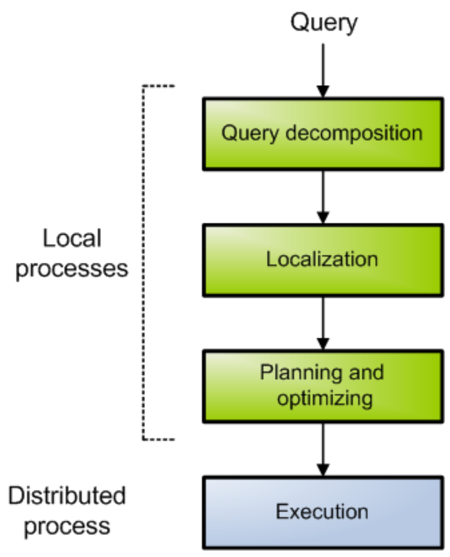


Figure 2.1: Steps in distributed query processing

The first action is to decompose the query into a tree representation in algebraic form. This step is done without knowledge of data localization, and is the same as in centralized databases. After forming the algebraic tree, it is restructured using heuristics to form a better tree, which means the worst execution plans are avoided. The next step is localization of the requested data, which means finding the sites that store the data, be it fragments or replicas. When data is localized, the optimizer searches for a good execution plan. The space of possible query plans is much larger than that of centralized systems. An additional problem is that maintaining relevant statistics for the query optimizer is more expensive, and the communication overhead between sites has to be accounted for. Most query processor use heuristics and two-phase optimizers to avoid evaluating every possible plan, sometimes at the expense of optimality [3]. A good execution strategy is found by evaluating multiple permutations of the restructured query tree according to a cost function. The cost function covers areas such as the amount of requested data at each site, the probability of matching when joining and network latency. The query is then executed in a distributed manner and the answer is sent back to the requesting site.

When it comes to concurrency, keeping the integrity of a distributed database is difficult. There are numerous approaches, each with varying degree of performance loss due to overhead. Some introduce the possibility of distributed deadlocks. When a site becomes unavailable due to a crash or network failure, the distributed database must continue its operation in a graceful manner. This takes a toll on performance, and is especially hard when the network divides into two or more partitions. Another challenge is the security aspect of distributed databases. Data sent across a network are liable to be intercepted.

Traditionally, the distributed database has not been very successful. This is due to several facts, like the advent of data warehousing and a lack of investments, though the most prominent is the failure to scale.

Adapting Database Requirements

Distributed databases suffer when facing network partitioning. This is because they are based on the fundamental idea of transactional consistency. Transactional consistency (ACID[1]) is one of the most important concepts in database theory. In distributed databases however, it is not possible to guarantee ACID combined with availability and tolerance of partitioning. In this context, high availability means that data is always available from some replica. Tolerance of partitioning means the entire system can sustain a partition between replicas. The CAP Principle (Consistency, Availability and Partitioning resilience) states that a distributed system can only enjoy two of those properties simultaneously [4].

[1]ACID stands for Atomicity, Consistency, Isolation, and Durability. Without these, the integrity of the database cannot be guaranteed. Atomicity refers to the ability to guarantee that either all of the elements of a transaction are performed or none of them are. Consistency refers to the database being in a legal state when the transaction begins and when it ends. Isolation refers to the ability to make operations in a transaction appear isolated from all other operations. Durability refers to the guarantee that once a transaction is committed, the effects will persist.

This means that the consistency requirement must be relaxed in order to attain high availability and tolerance for partitions. Partitioning is unavoidable in any distributed system beyond a certain scale.

2.2 Object Relational Databases

This section briefly presents the relational model and explains the concept of object relational databases [1, 5].

The Relational Model

Relational databases came into existence in the late 1970s and provided a robust foundation for the storage, retrieval and manipulation of data [1]. The relational model is based on the mathematical idea of a relation, which is represented as tables in a database. Relations are used to hold information about entities, and are implemented as two-dimensional arrays. The rows in the array, called tuples, refer to specific instances of objects, and the columns in the array correspond to attributes. Each attribute has a domain, which is a specific type of value. See figure 2.2 for an example database.

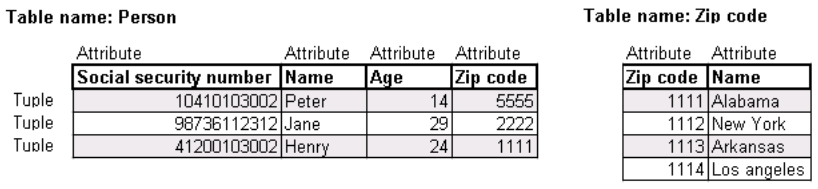


Table name: Person

	Attribute	Attribute	Attribute	Attribute
	Social security number	**Name**	**Age**	**Zip code**
Tuple	10410103002	Peter	14	5555
Tuple	98736112312	Jane	29	2222
Tuple	41200103002	Henry	24	1111

Table name: Zip code

Attribute	Attribute
Zip code	**Name**
1111	Alabama
1112	New York
1113	Arkansas
1114	Los angeles

Figure 2.2: Tables in the relational model

The relational model works well for processing typical business data, but it falls short when it comes to storing and retrieving complex data such as diagrams, geographical data or images. There is also a semantic gap between the relational model and object oriented programming languages, that means there has to be a conversion between the two whenever they are used together. Object oriented databases were pitched as the solution to these problems. They can store complex data and can incorporate specialized access methods. Hence, vendors of relational DBMS have extended the relational model with various object oriented capabilities to comply with the emerging demands. Unfortunately, there is no single, precise definition of an object relation model, as there are many different approaches and features in each specific extension.

The Object Relational Model

The most important features of the extended models are abstract data types, collections, inheritance, polymorphism and functions.

- **Abstract data types**
  An abstract data type (ADT) can be used for creating complex abstractions of objects, providing a simplified interface for querying and manipulation. For example, when storing geometric objects, representing them as ADTs eases the implementation of functions such as intersect and area. The possibility of incorporating specialized index structures is also an advantage.

- **Collections**
  Having collections in a object relational database means multiple values can be stored in a single column of a table. Examples of collections are lists, sets and arrays.

- **Inheritance**
  Inheritance makes it easier to model complex structures and relationships. It enables the use of polymorphism, which is a powerful construct in object orientation that enables an object to be treated as a member of multiple classes.

- **Functions**
  Functions can manipulate data in the database without the need for client side code. Centralizing the implementation this way makes for easier maintenance of code but increases the load on the database server.

There are several advantages of extending the relational database model as opposed to using a purely object oriented approach [5]. First of all, many companies have invested significant funds in relational databases, making it attractive to build on what they already have. Second, purely object oriented databases can currently not compete in terms of performance. Third, the relational model can be kept while the benefits of object orientation are available.

Third, it enables the centralization of functionality in which complex operations can be performed centrally.

2.3 Peer-to-Peer Systems

A peer-to-peer (P2P) system [6] is a distributed system where participating nodes are treated as equal and are able to act as service providers as well as consumers. This is in contrast to the traditional client-server architecture, where a few servers provide services for their clients. The decentralized structure gives a higher utilization of each node's resources, and P2P systems are thus very scalable.

Nodes in a P2P system must act autonomously due to the lack of central coordination. This is an advantage when it comes to robustness since there is no single point of failure. It is however a challenge when it comes to resource localization since the nodes do not have a complete and consistent perception of the system's global state. The following sections will present different approaches to localization in P2P systems.

2.3.1 Resource Localization

This section contains an overview of three different strategies for indexing and searching for data in P2P systems. These are the centralized, the decentralized and the structured approaches. Figure 2.3 illustrates their characteristics.

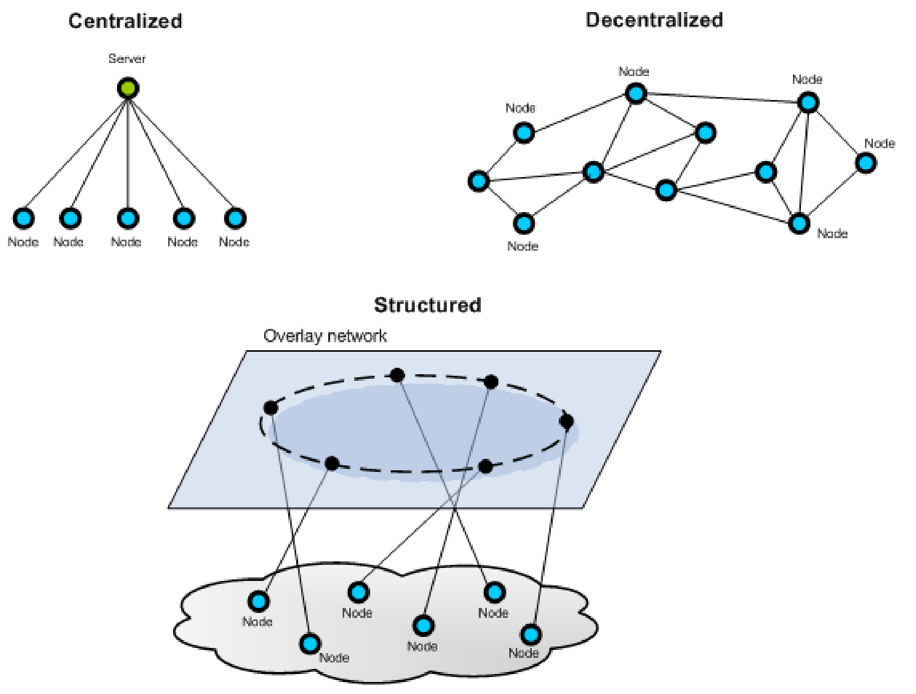


Figure 2.3: Different Peer-to-Peer approaches

Centralized (Hybrid) P2P Systems

Centralized P2P systems rely on a centralized entity that indexes all the data in the network. Nodes connect to the network through the centralized entity, and in doing so, their data is registered in the index. When a node leaves, its data is removed from the index. Any of the connected nodes can search for data by querying the centralized entity. Thus, the index acts as a mediator between the clients in the system, telling them where they can find data. This is a hybrid model where the only P2P interactions are file transfers from one peer to another. The main disadvantage is that the centralized entity is a bottleneck that restricts the scalability of the system. It also constitutes a single point of failure.

Decentralized P2P Systems

In order to obtain better scalability and fault-tolerance, the functionality of a centralized index can be shared among the nodes in the network. This is the concept of the decentralized P2P system, such as the Gnutella protocol. Rather than having a few of the nodes know the state of the network, this information is stored among all the nodes. However, it is not feasible to construct fully interconnected systems without losing scalability. This would require each node to maintain an enormous amount of state when the number of nodes grows. The solution is for each node to have connections to a small number of peers[2]. These nodes may be connected to other nodes, thus creating a loosely connected network. Lookups are performed by flooding requests to all peers, which in turn pass the request on to their peers and so on. This approach generates a large amount of network traffic, so each request is given a time to live (TTL)[3]. The TTL prevents queries from reaching further than a limited horizon. Because of the horizon, one cannot guarantee that a lookup is successful even if the requested resource exists somewhere in the network. It has been shown that a 18 byte request in a Gnutella network can produce hundreds of megabytes of network traffic [7].

Structured P2P Systems

Structured P2P systems organize the connected nodes in a overlay network to obtain decentralization. The purpose of the network is to improve the search procedure so that requests are routed rather than flooded. The most common overlay network used in P2P systems are distributed hash tables (DHT). The next section contains a thorough discussion of DHTs.

2.3.2 Distributed Hash Tables

A DHT is a virtual network on top of the underlying physical network. Two adjacent nodes in the overlay network may be far away from each other in terms of other distance metrics (geographic, number of IP hops, round-trip time). As the name implies, a DHT is also a lookup mechanism for P2P systems which replaces the flooding approach used to spread requests in decentralized systems. The basic idea is to route requests in the direction of the requested data. This is done by partitioning the identifier space among the nodes in the system. Each node is made responsible for its designated part of the identifier space. This can be seen as a hash table where each node acts as a hash bucket. Values are mapped to the identifier space by assigning a key to them, generated by applying one or more hash functions to the value.

DHT implementations are usually expanded to include functionality for inserting and retrieving values. These functions use the lookup function to find the node which is responsible for the value's key.

The rest of this section presents different aspects of DHT implementations.

[2]Gnutella nodes usually have connections to 4 other nodes

[3]In Gnutella, the TTL is typically set to 7 hops

Topologies

A DHT's topology defines how nodes are connected, how the identifier space is partitioned and how lookups are performed. Several different topologies have been proposed for DHTs [8]. Here is a presentation of the most common ones:

- **Ring topology**
  The ring topology (figure 2.4) uses a cyclic one-dimensional identifier space. The distance between two keys A and B is $(A - B) \bmod N$, where N is the size of the identifier space. Each node is given a key[4] and hence assigned a position in the ring. A node is responsible for the keys in the range between the preceding node in the ring and itself.

  The requirement for the ring to work properly is that each node maintains a connection to its successor. This would, however, imply linear lookup-times and that the ring is broken if a single node fails. In order to increase the fault tolerance each node is maintaining a list of successors with k nodes. This means that k nodes with consecutive identifiers must fail to break the ring. Logarithmic lookup-times are achieved by introducing a finger-table at each node. Node n's finger-table maintains connections to the nodes with keys $n + 2^{i-1}$, where $1 \leq i \leq m$ and m is the length of the identifiers.

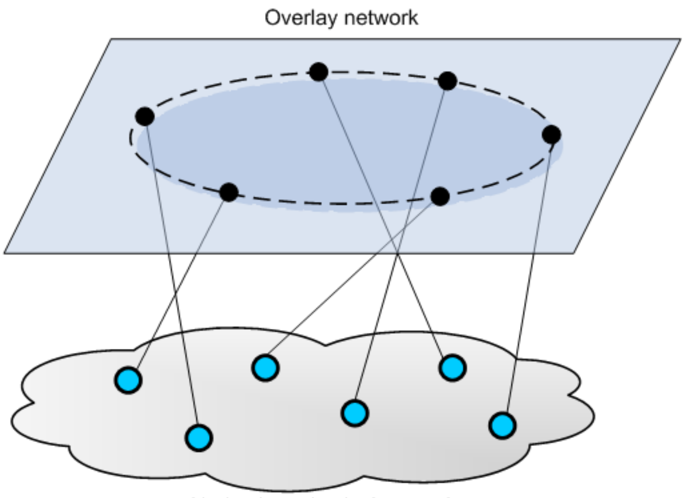


Figure 2.4: Ring topology

- **Tree topology**
  The tree topology (figure 2.5) organizes the identifier space in a hierarchical fashion. The distance between two nodes is the height of their smallest common sub-tree. Each node maintains a routing table with connections

[4]Node identifiers can be created in a number of ways. For example by hashing the node's IP address and port number or it can be issued by a centralized authority.

to $(d-1) * \log_d N$ nodes where d is the tree's fan-out[5] [6]. This gives each node a certain degree of freedom when it comes to selecting routing-table entries.

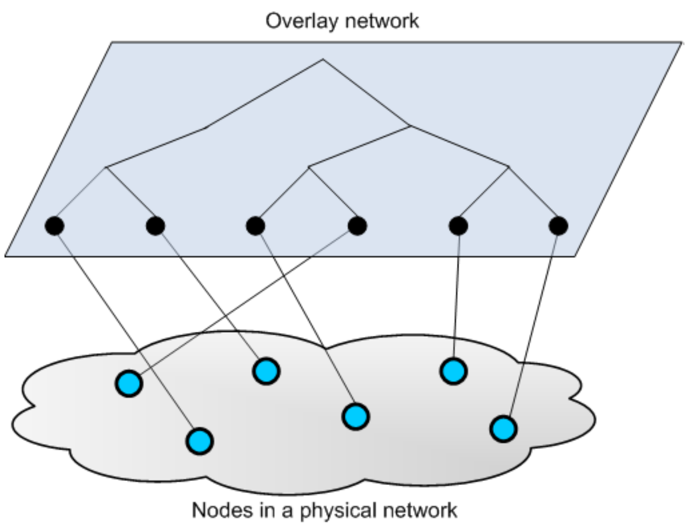


Figure 2.5: Tree topology

- **Hypercube topology**

  Hypercube topologies (figure 2.6) use a coordinate system shaped like a d-torus[7]. The identifier space is partitioned into zones. Two nodes are neighbors if their coordinate spans abut along one dimension. The path-length between two nodes is $O(\sqrt[d]{N})$. An advantage of hypercube topologies is that there exists, dependant on d, a number of alternative paths between two nodes. This can increase the network utilization.

Lookup

A lookup is an operation that finds the node responsible for any given value. It consists of routing lookup messages towards the node that is responsible for a given key. At each step of the process the message is sent to the node that is closest to the target. Lookup-operations can be divided in to two sub-categories, iterative (figure 2.7[8].) and recursive (figure 2.8[9]). Both approaches have advantages and disadvantages.

Iterative lookup is performed by returning a response to each lookup-message to the sender and thus giving the initiating node full control over the process. The downside to this approach is that the number of messages required to reach

[5]A tree's fan-out refers to the number of child nodes each node in the tree may have.
[6]For every of the $\log_d N$ levels in the hierarchy, each node needs to maintain connections to the $d-1$ other sub-trees.
[7]A d-torus is a d-dimensional Cartesian space that wraps around the edges
[8]Adopted from [9]
[9]Adopted from [9]

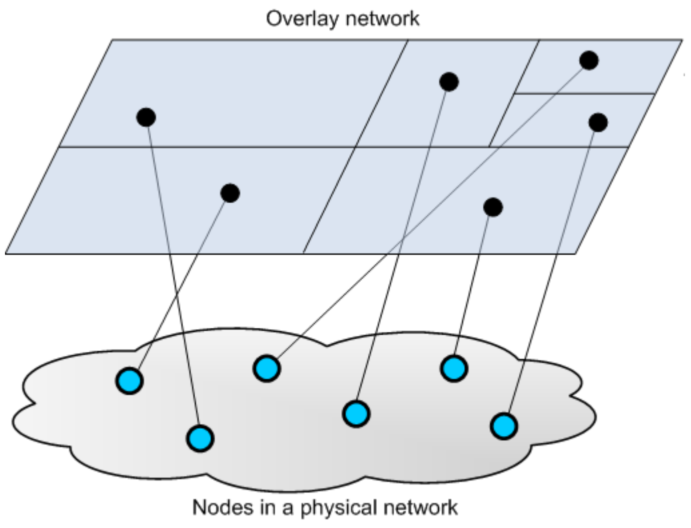


Figure 2.6: 2-dimensional hypercube topology

a node is somewhat increased[10]. It also requires that a lot of TCP-connections must be initiated or that an alternative protocol is being used.

Recursive lookup is performed by passing on lookup-messages to the next node without involving the initiator. This approach does not give the initiator any control over the process or information about why a lookup fails (there could be a malicious or overloaded node somewhere along the path). Possible solutions to this problem are using a probabilistic routing scheme[11] [11] instead of a deterministic one, or switching to iterative lookup after a given number of attempts [10]. Recursive lookup uses the connections that are already established between the nodes in the network, but one might want to establish a connection to return the result directly from the target node to the initiator since this will decrease the required number of messages.

As seen in the topologies section, most DHTs are capable of performing lookups with $O(\log N)$ messages. One should be aware that this is not equal to the number of IP hops. A lookup request from a computer in Trondheim to one in Oslo could very well visit every continent in the world before completing if routing table entries are not carefully selected.

Adapting to changes

In P2P systems there is a continuing process of adapting the network as nodes join or leave. The continual process of nodes joining and leaving is often referred to as *churn* [9]. Nodes join and leave as they please and therefore the network needs a coping strategy that does not disrupt the operation of the system.

[10]The number of messages is increased by a factor of 0.6 according to [10]

[11]In a probabilistic routing scheme messages will only be forwarded to the best node with a certain probability. If a lookup fails, chances are that the lookup will follow a slightly different path the next time it is retried, and it will eventually succeed.

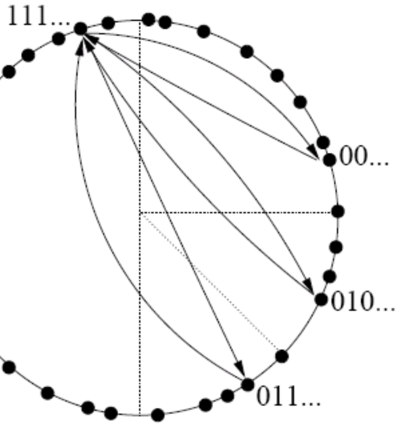


Figure 2.7: Iterative lookup.

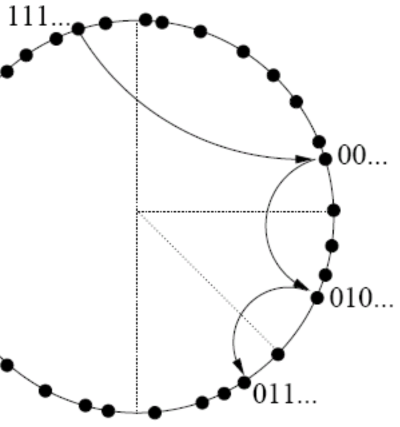


Figure 2.8: Recursive lookup.

When a node wants to join a network, it may connect through any node that is already in the network[12]. The new node is then given a share of the identifier space (or it may share the space with another node) and needs to acquire the keys belonging to that share. It also needs to build its own routing table and tell other nodes to update theirs.

A node may intentionally leave the network or abruptly leave due to failure. In the first case the node may notify other nodes and give away its keys before leaving and thus leaving the system in a graceful manner. In the case of failure, the other nodes in the network need to be able to detect the node's departure, update their routing tables and take over the failing node's share of the identifier space[13]. The updates can be done in a reactive or a periodic manner [9]. Doing

[12]The process of connecting to a P2P network is called bootstrapping. The node used to enter the network is often referred to as a bootstrap node or boot node

[13]In many DHTs, such as Chord, this happens as a consequence of the topology and no

it in a reactive manner means that a join or leave triggers the update process. Periodic means that updating is done as a continuous process.

Reactive updates ensure that the system's state is kept as consistent as possible and that the number of messages passed around is kept at a minimum. However, in the event of multiple nodes joining or leaving, this approach may trigger an immense amount of network traffic. This may be enough to cause even more nodes to fail and thus create even more update processes.

Periodic updates generate a constant rate of traffic and have a less probability of keeping the routing-tables in a consistent state. However, routing performance does not suffer much from small inconsistencies. DHTs with periodic updates are more robust against churn [9].

Searching in DHTs

It is difficult to perform effective searches or range queries in P2P systems organized using DHTs, since they only provide lookup on exact key matches. In order to be able to search in such a system one must either flood queries like in Gnutella or build another index on top of the DHT. Other alternatives including using tree structured DHTs or sharing routing information have been proposed as well [12, 13].

2.4 Previous Work

The PORDaS project of fall 2005 resulted in a experimental prototype of a P2P database and outlines for further work. In order to keep this master thesis self-contained, the following section will present a brief summary of that project. An analysis of the scope and direction of the thesis is presented in section 3.1.

PORDaS is a system where independent database systems are connected to a DHT to be able to share and access data from other repositories. The system is location transparent and based on resource identifiers. The DHT is used to form a global shared memory where an index is stored. The index entries are at the schema-level and contains mappings between resource and node identifiers.

PORDaS also contains a metadata index that makes it possible to search for resources based on assigned keywords. An entry in the keyword index contains a mapping between the keyword and all related resources. It is not necessary to put keywords in the DHT index, but it was originally added as an initial exercise and was later kept for convenience. An alternative would have been to create a separate structure for the keywords, for instance formed as a hierarchy of terms. This type of indexing is not optimal since there is only a limited amount of possible keywords and their popularity is not uniformly distributed.

The indices are maintained using soft state and can thus never guarantee any form of consistency. Each index entry is given a time to live which is periodically updated by its owner. The advantage of this apporach is that it only requires

effort to repartition the identifier space is needed a limited amount of resources to keep the index fairly up to date. Soft state indices are also quite robust when facing a large number of node failures [14].

PORDaS' query processing capabilities are limited, with selection and projection as the only available operators. One of the main goals of this thesis is to expand the query processor to include join-operators as well.

The software is written entirely in Java. This choice was made to allow rapid development of the prototype. The program has been designed with modularity in mind. The DHT, local storage is separated from the rest of the system and can easily be replaced[14]. PORDaS is only a middleware layer itself and can therefore be included in any application. The DHT used is FreePastry [15], an implementation of Pastry [16] (Appendix A) written in Java. The storage layer was initially an embedded version of MySQL [17] (not Java), but was later replaced with Derby [18] (Java) to ensure compatibility with all available resources[15].

A test application was built to run experiments with PORDaS on clusters. The experiments showed that the system behaved as expected, but needed a scheduling mechanism to ensure that nodes can only start a limited number of transfers at the same time. FreePastry is a best-effort service and does not retransmit messages lost due to overflow in the message buffer. This caused queries to fail and response times to increase exponentially when nodes received queries at a higher rate than they could be processed.

2.5 Related Work

PORDaS is in many ways related to the fields of distributed databases [2, 3] and distributed query processing [19].

PORDaS is similar to the PIER project [14, 20], which is a distributed query engine built as middleware to create other applications over. Its prime purpose is scalability, which is achieved through the use of a DHT structured overlay network. The data in PIER are managed locally at each site, and is structured using a relational model. Transactional consistency is relaxed in order to attain the goal of massive scalability and resilience to network partitioning. PIER differs from PORDaS in that it indexes all tuples in the system, while PORDaS maintains an index at the table level. This means that PIER has a larger index. Additionally, PORDaS focuses on increasing availabilty and performance through replication.

ObjectGlobe [21] is a distributed query processing infrastructure, where sites in a network offer their data and processing capabilities for a fee. It relies on a centralized lookup service and focuses on security and privacy issues. Mariposa [22] is a system similar to ObjectGlobe. It employs an economical model to drive a distributed database. The idea is to have sites buy and sell data processing services using a virtual currency. Where queries are processed depends not only on data location but on which node is selling its processing power.

[14]Replacing the storage component or DHT requires no changes to PORDaS. The new component must adhere to the storage or network interface.

[15]MySQL 5.0 was released later on with improved platform support

An introduction to queries in the context of P2P networks with a DHT overlay network is given in [23]. A data model specifically designed for P2P applications is introduced in [24]. The idea is to allow for inconsistent databases and to allow for interoperability in the absence of a global schema.

AmbientDB [25] is a P2P query processing architecture using the relational datamodel. A DHT is used to connect clients and to create indices that support transparent query processing in an ad-hoc P2P network.

PeerDB [26] takes a different approach by using agents to collect data in a unstructured P2P database system. The clients can reconfigure their network tables in order to obtain better performance. Additionally, data can be shared without a shared schema. This is done using a keyword thesaurus, where each relation in the system is associated with a set of keywords.

Chapter 3

Analysis

This chapter defines and analyses the fields of interest for the master thesis. The chapter opens with a section on scope, where the boundaries for the thesis are defined. The next parts revolve around the core subjects; the study of a distributed query processor and ways to optimize the performance of a P2P database system. The analysis ends with a section on considerations related to performance tests and experiments.

3.1 Project Scope

This section presents the scope of the master thesis. It gives an overview of the main areas of interest, and it defines the boundary of constraint.

The core of the thesis is distributed query processing, and how this can be done sensibly in a P2P environment. An important question in this respect is how to locate and retrieve resources without global knowledge of the state of the system nor which peers store which data. This challenge, and the previously mentioned goal of robustness, is met with the use of an overlay network. In short, the use of a distributed index is studied. Another question is how the resources in the system can be utilized in an efficient fashion when performing queries. The possibility of pipelining and parallelizing the query processing is examined. A secondary area of focus is to search for ways to increase the performance. This work contains the use of replication, caching fragmentation and the use of specialized indices.

In order to focus on the core subjects of the thesis there are several subjects related to P2P systems and databases that will be left out or simplified.

The query language will not be extended further than adding join operators. This means that functions such as nested queries, aggregations and unions will not be supported. The object relational capabilities of the previous PORDaS project will not be expanded. This means that there will still be abstract data types, but no effort will be made to include additional features such as inheritance, polymorphism, collections, functions or other of the features seen in modern ORDBMSs.

The facilities for navigating through or finding schema definitions are very limited in PORDaS. Currently, schema definitions in PORDaS are identified by their hash value. This is not a very user friendly solution and PORDaS could possibly benefit from a solution similar to XML namespaces where every element within a namespace must be unique. Another similar improvement is better categorization of schemas by using an ontology and semantic webs. This could allow for a better keyword search. These issues will however remain simplified.

Security issues will not be considered. Hence there will not be any measurements to prevent denial-of-service attacks or general abuse. Nor will there be any focus on trust or solving the problem of free-riders. It is assumed that users are benign, trustworthy and willing to contribute.

3.2 Distributed Query Processing

This section contains an analysis of distributed query processing in a P2P database. It begins with a presentation of how a traditional query processor with a static query optimizer could be adapted to a widely distributed environment [2]. Finally, a more dynamic query processor is presented.

3.2.1 Traditional Distributed Query Processing

This section analyses the adaptation of a traditional distributed query processing theory to a P2P setting, and identifies problems that may arise.

Query decomposition

The first step of any type of query processing is query decomposition. The goal is to convert the query from a textual representation into a relational algebra tree. The structure of the tree decides the order of execution in the execution step.

Before this happens, syntax errors are discovered and malformed queries are rejected by the parser. This step also removes redundant expressions. Figure 3.1 gives an example of a conversion from a textual query to its equivalent algebra tree. Basically, all the projections in the query will be set as the root of the tree, followed by any selections. Finally, cartesian products are put in the bottom of the tree.

An algebra tree has many equivalents, some of which are better in terms of execution performance. The last part of decomposition is therefore restructuring of the initial algebra tree to avoid the worst solutions. The restructuring is done using a set of conversion rules [2] on the initial tree. The basic idea is to reduce the size of the intermediate results as early as possible using heuristics. This

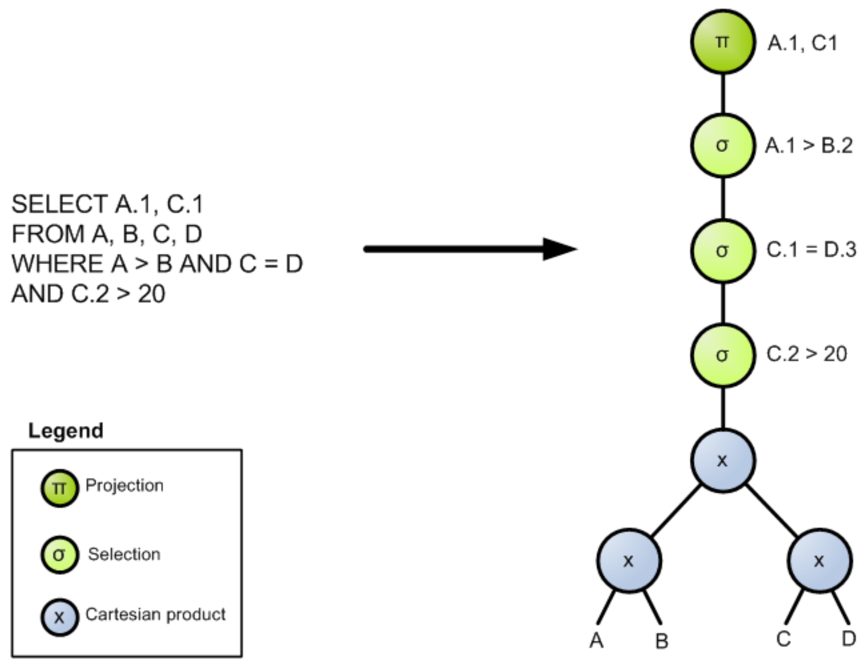


Figure 3.1: Decomposing a SQL query into an algebra tree

translates to smaller inputs to the most expensive operators and less data shipping between sites. Restructuring is done by pushing selections and projections towards the leaves in the tree. Cartesian products are avoided, and if they can't be, they are pulled towards the root. Figure 3.2 gives an example on how a tree can be improved. Two of the cartesian products in the tree have been converted into joins by consuming the selections in the tree, selections and projections are pushed down.

Data localization

The next step is data localization. First, the schema definition for each of the tables in the query must be retrieved. The schema definition is needed in order to type-check the attributes and to check that the resources referenced in the query actually exist. Second, the nodes which store the data involved in the query must be identified. If any of the tables are replicated, these must be handled appropriately to avoid duplicates in the result. In the case where a DHT is used, an index can be distributed among the nodes. This index can contain information that connects schema definitions and a list of the nodes that store them. The schema definitions in the index must be uniquely addressable. This can be done through the use of a hash function. Queries can then reference specific tables by using hashed schema definitions as identifiers. For ease of use, aliases can be made to make the queries easier to read and create.

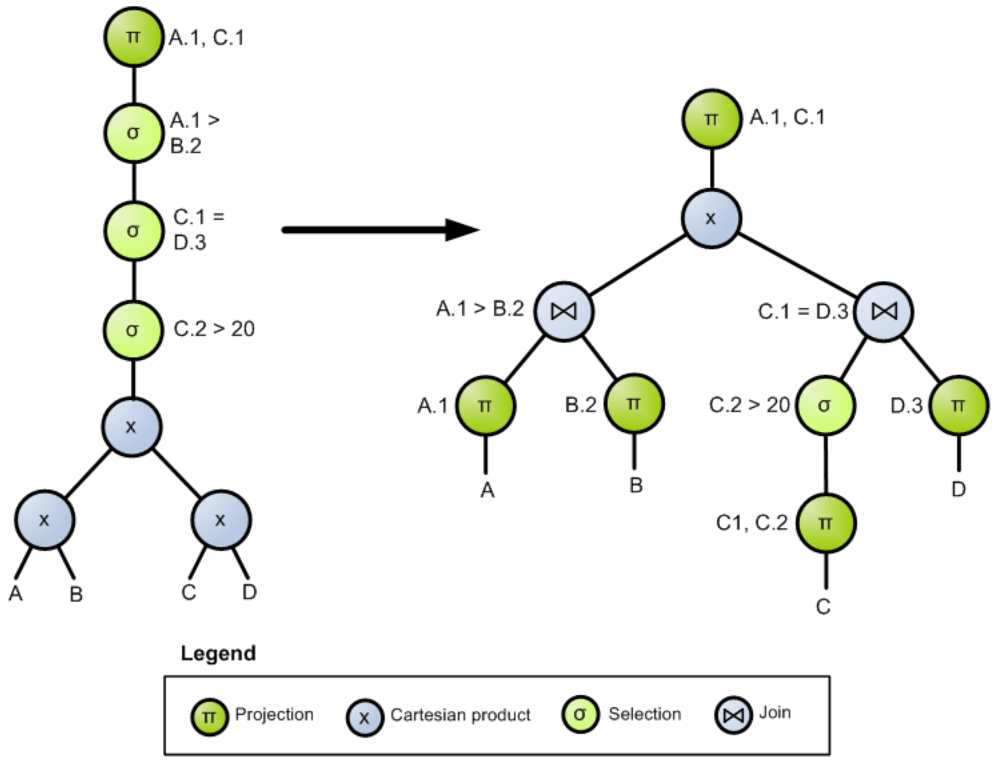


Figure 3.2: Restructuring an initial algebra tree

Planning and optimization

After the localization step comes planning and optimization. The output of this part of the process is an optimized execution plan that decides how the query tree will be executed, and which sites are to be involved.

The optimized plan is found by first defining the search space for the query. As the search space increases dramatically in size with the number of tables and operators involved (optimization is an NP-hard problem), considering all the options is too expensive. Instead, heuristics are used to find a good solution within a feasible amount of time. The search space is defined by applying transformation rules to the query tree, thus creating a set of equivalents. For instance, many alternatives are made by permuting the join order as it has a significant effect on the performance of the query. Additionally, the search space can be constrained by restricting the shape of the tree. The disadvantage is that in disregarding certain tree structures, the most efficient strategy may be pruned away. There are two classes of trees that are important in this respect, linear and bushy trees. The binary operators in a linear tree must have at least one operand that is a base relation. The operators in a bushy tree do not have this constraint. It is obvious that bushy trees offer more possibilities for executing queries in parallel in a distributed setting. Figure 3.3 illustrates the difference between the two classes of trees. The two cartesian products in the bushy tree could be resolved concurrently by different sites, but this is not possible with the linear tree.

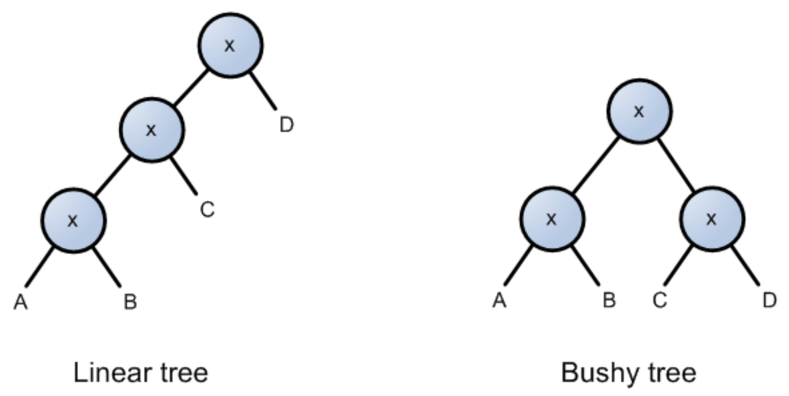


Figure 3.3: Classes of algebra trees

The choice of execution strategy is another parameter to consider when creating the search space. For instance, there are several choices on how to perform joins [27]. Nested loop, sort-merge, hash-join and hybrid hashing are some of the procedures used when the queries are local. Some of these have been adapted to fit the requirements of distributed databases. These will be further explained in the section on the execution phase.

When the search space has been established, a cost function is used to score each of the alternatives. In centralized query processing [28], the search space is bounded by the local database, but in the decentralized case, the network has to be accounted for. The cost function itself is based on a cost model, which predicts the costs of operators and the size of results. The more accurate the model and the knowledge of the input data is, the less the chance is of creating a bad execution strategy. A weak cost function can be damaging to the response times and the throughput of the system. The inputs to the cost function are statistics collected about the state of the system. This includes table cardinalities, the minimum and maximum values for numeric attributes, the presence of indices, the type of indices and so on. The cost of network communication can be approximated by a constant, or it can be more precisely determined by storing the average latency for each site. In a P2P network, the connected computers are likely to run on hardware with different performance. The cost function has to deal with this fact, either by getting information about the performance or assume it is the same at all sites. Such a simplification narrows the search space considerably.

It is hard to make realistic assumptions about the state of the network, especially in a P2P environment. The nodes in the network may have different types of connections, the currently connected set of nodes may be in flux and the load at each node may differ from one moment to the next. By assuming constant communication cost, the complexity of the cost function is greatly reduced.

At any rate, the statistics must be accessible from somewhere in the distributed system, and they need to be maintained. In general, the more precise the statistics are, the more expensive they are to maintain. In a P2P setting, having each node keep a repository of the global state is prohibitively expensive if the goals of scalability are to be met. One possibility is to distribute this information as an index among the nodes by using the DHT. However, this is best when the data are rarely updated. If not, the optimizer would operate on stale information, disrupting its performance. Additionally, maintaining an index of statistics requires a certain amount of resources.

An example cost function for distributed query processing is presented in [2]:

$T_{total} = T_{CPU} * \sharp insts + T_{I/0} * \sharp I/Os + T_{MSG} * \sharp msgs + T_{TR} * \sharp bytes$

$T_{CPU} * \sharp insts$ is the time taken to process all instructions in the CPU. $T_{I/0} * \sharp I/Os$ is the time needed to perform I/O. The rest of the formula, $T_{MSG} * \sharp msgs + T_{TR} * \sharp bytes$, is the cost of communication, where a constant latency is assumed. T_{MSG} is the constant time taken to start and receive a message and $\sharp msgs$ is the number of messages sent. $T_{TR} * \sharp bytes$ is the constant time taken to transfer $\sharp bytes$. In a P2P network with a DHT, the time taken to look up resources should be added to the formula.

Using a static optimizer in a P2P database is hard because of fluctuations in the connected population and the fact that loads are likely to change during a query. This makes it difficult to assess the state of the system through statistics, which makes the optimizer base its decisions on uncertain data.

Execution

The final step in query processing is to execute the plan created by the optimizer.

When transferring data in a P2P database, it is beneficial to pipeline tuples to maximize resource usage [1]. Pipelining means that tuples are processed when they arrive and sent as soon as they are created. This reduces performance penalties du to skew[1] since operators do not have to wait until they have received all tuples from their operands. This way, the result is received as a stream of tuples, adding the benefit that the first tuple in the result can be seen earlier. Another advantage is that there is no need for storing intermediate results in a temporary files, as they are shipped off immediately. The granularity of the pipeline can be varied. Sending blocks of tuples instead of single tuples will save message overhead, but will require intermediate storage. Finding the right block size is a trade-off between the two.

In the case of errors in the execution phase, there has to be a mechanism for coping with failure. Errors can be caused by many reasons, for example a site can have failed, parts of the network may be down or the load at a site may be too large. There are several appropriate actions that can be taken in the face of an error. Different strategies exist such as aborting the query, restarting the query, re-planning it or merely delivering the partial results. The choice of strategy depends on the requirements of the application.

The abort and restart mechanisms are straightforward. The query execution halts and in the case of restart is resubmitted to the query processor. Both

[1]Skew refers to an uneven delivery of results from the operators involved in the resolution of a query.

methods are easy to implement, but they risk discarding useful partial results. Re-planning means that the query is re-planned to avoid the failure. Either a new plan is submitted, or if it is possible, the part of the plan that failed can be re-planned. Re-planning means that partial results can be kept while a new plan is made for the remainder of the query. This strategy could be more effective, but it is also more complex. The alternative of delivering only partial results ensures that the response time is somewhat unaffected by failures, but the results suffers from being incomplete.

Performing joins in a P2P database where the network is volatile, is not an easy task. On the bright side, the distributed nature of P2P systems lends itself to traditional parallel join algorithms. An attempt was made to incorporate these in PIER [14], which is a distributed query engine using overlay networks. The approach taken was to implement two binary equi-join algorithms based on adaptation of parallel and distributed schemes. Suggestions are also made for optimizing the schemes.

One is a DHT-based version of the *pipelining hash join* [29]. The concept is shown in figure 3.4 which illustrates a selection between to tables stored in different namespaces in a DHT. The nodes in the DHT that have data from one of the relations to be joined, scan their data locally. This data is then rehashed into a new namespace based on the join key. The relations are then probed, and the matching tuples are sent to the next join step. If it is the last step, they are returned to the node which initiated the query.

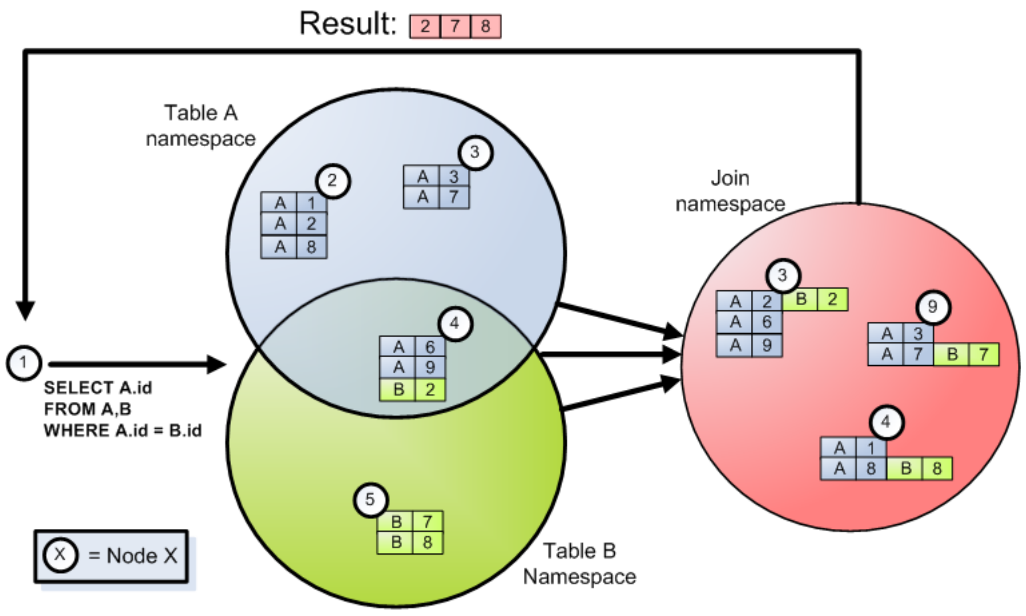


Figure 3.4: Pipelining hash join in a DHT

Another approach, namely *fetch matches*, is a variant of a traditional distributed join algorithm, that only works when one of the relations is already hashed on the join attributes. Basically, the method saves the work of rehashing both relations. All the nodes that store the other relation, go through all their tuples in this relation and performs a get on the hashed relation for each tuple. The results are forwarded to the node which initiated the query.

Optimizations of the schemes above include the use of *bloom filters*, which have the effect of minimizing the total bandwidth consumption. When performing a join, every node storing either of the relations to be joined, compute a *bloom filter* for each of the relations. These filters are then distributed to a small temporary namespace, where the filters are OR-ed together and then dispatched to the nodes holding the opposite table. The *bloom filters* are then used to minimize the amount of tuples which are processed for the distributed join outlined above.

Another optimization is the *symmetric semi-join*, which saves bandwidth by avoiding rehashing both the joining tables. The idea is to limit the initial communication by projecting each of the tables locally to their primary keys and join keys, and then doing a symmetric hash join on the projections as explained above. The results are sent into *fetch matches joins* on each of the tables' primary keys.

3.2.2 Adaptive Query Processing With Eddies

Due to the problems associated with using a static query optimizer in a volatile environment it is necessary to consider an alternative that avoids the trouble of pre-optimizing and having a metadata catalogue altogether.

An eddy [30] is a tuple router which is interposed between the data processed and are eventually routed as output. Eddies employ a routing policy to decide where a tuple should be routed next. If the routing policy responds to feedback from the operators it can adjust the flow of tuples during the execution. This allows eddies to dynamically change query plans during execution and obviates the need for an optimizer or a metadata catalogue. Figure 3.5 shows the difference between a traditional query plan and a query plan with an eddy.

The eddy was originally designed for centralized query processing but the concept has also been adapted to distributed query processing as Federated Eddies (Freddies) [31]. Freddies are designed to work within PIER [20] and will serve as an example since they are the only known implementation. Other proposals for eddies in distributed query processing has been made in [32, 33].

Freddies are quite similar to eddies with a unique instance of a freddy on each node participating in a query. In addition to routing tuples to its local operators, freddies has the option to put tuples in the DHT. Figure 3.6[2] shows a freddy instance executing a query plan using hash ripple join. The freddy may route to local operators or rehash tuples to other nodes.

The first step of freddy-based query processing is to create a query plan where cross products are avoided. The query plan contains an operator graph where acceptable orderings encoded. The same query plan is disseminated to each node participating in the query.

[2]Adopted from [31]

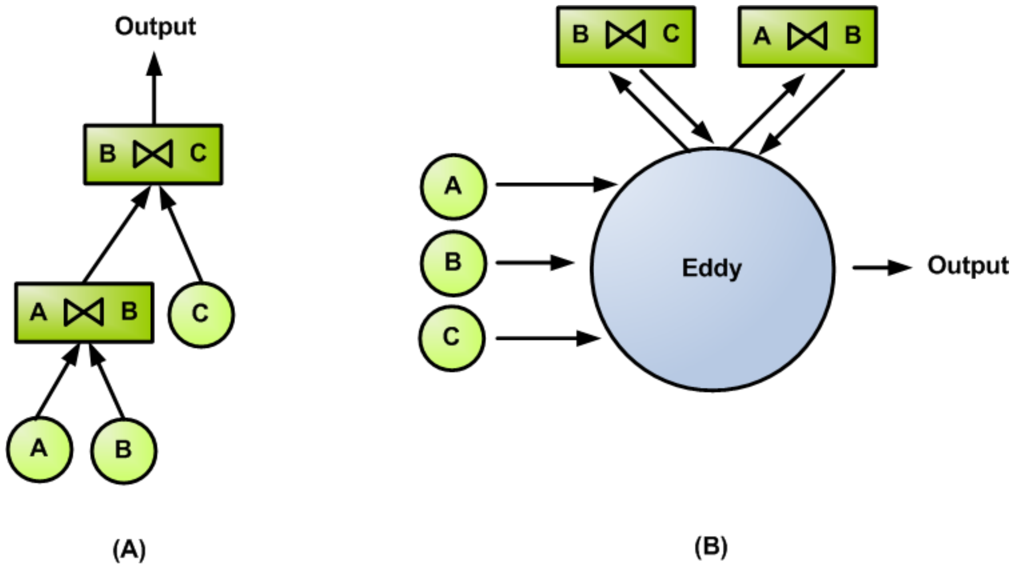


Figure 3.5: (A): A traditional query tree. (B): Query execution using an eddy.

Each tuple carries two bitvectors, *done* and *ready*, indicating which operators it has been processed by and which operators it is eligible to be processed by. Each time a tuple is outputted from an operator it has the appropriate *done* and *ready* bits set. If the output is the result of a binary operator the vectors from both input tuples are ORed. When a tuple has all its *done* bits set it is outputted.

The performance of eddies relies on their routing policy and the ability to make the correct assumptions about which join algorithms to use. These choices have to be made on the basis of heuristics and with the constraint that each relation does only participate in one join. This prevents eddies from exploring the entire space of valid query trees since all that can be done during the execution phase is to alternate the ordering of the operators[3].

In order to enable reordering of operators one must be able to change the query plan while preserving its state. To do so continuously requires operators that are *pipelined*[4] and avoids *synchronization barriers*[5] while having frequent *moments of symmetry*[6].

The ripple join [35] family has been proposed as suited join operators when using

[3]Solutions to some of the limitations of eddies has been proposed in [34]

[4]If the operators are not pipelined one will have to wait until an operator has completed processing before one can reorder the operators.

[5]A synchronization barrier is a structure that causes each thread in a collection of threads to block until the entire collection is blocked, at which time the entire collection is released. Join algorithms encounter synchronization barriers whenever they must wait for input. Skew can cause significant overhead if there are frequent synchronization barriers.

[6]A moment of symmetry is a point in execution where the ordering of inputs to a join can be modified without significant changes to the state of the operator. When execution reaches a synchronization barrier it ends the scheduling dependency between the inputs. Many join algorithms has a moment of symmetry at this point.

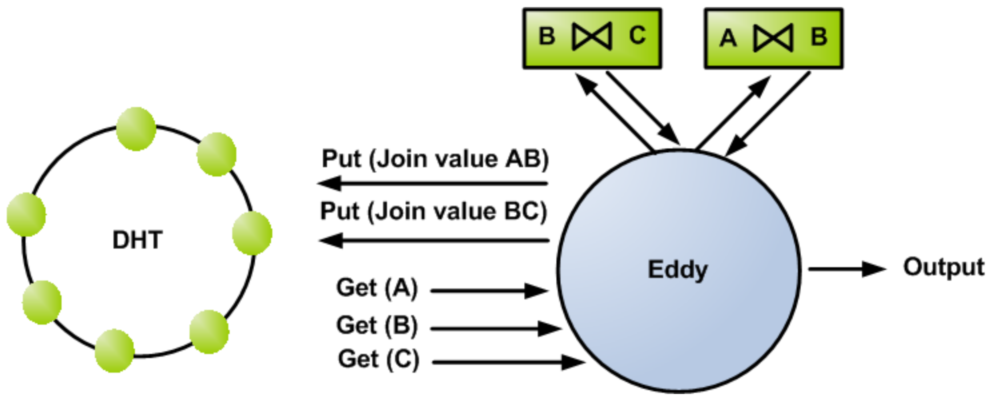


Figure 3.6: Query processing with a freddy

eddies[30, 31]. Index and hash ripple joins can be used for equi-joins and simple or block ripple joins can be used for non-equi-joins. These algorithms cannot compete with the best case performance of more traditional algorithms, but their adaptability make them favourable since best case scenarios are unlikely to exist for long periods of time in a volatile environment.

3.3 Optimizations

This section presents two promising areas for increasing the performance of distributed query processing. These are replication and having a specialized index to support range queries.

3.3.1 Replication

In a distributed system like PORDaS it is unlikely that the entire load of the network is uniformly distributed among the nodes. Nodes may contribute unevenly in terms of how much data they share and which resources they hold. Another factor is that the query distribution is more likely to follow a power law distribution, rather than a uniform distribution [36]. A distributed database is not scalable if it is unable to handle hot spots. In addition to having an uneven load distribution one must also be able to handle objects increasing or decreasing in popularity. An increase in popularity, either a slow change or a more sudden flash crowd[7] effect, requires that new replicas are made. When popularity decreases it is desirable to decrease the amount of replicas to avoid wasting resources.

[7]"Flash Crowd" was a 1973 short story by science fiction author Larry Niven, one of a series about the social consequence of inventing an instantaneous, practically free transfer booths that could take one anywhere on Earth in milliseconds. One consequence not predicted by the builders of the system, was that with the almost instantaneous reporting of newsworthy events, tens of thousands of people worldwide would flock to the scene of anything interesting. Source: Wikipedia

Another potential advantage of having replicas is the possibility of using replicas for parallelism. This could be done either by performing tasks in parallel or retrieving data from multiple sources simultaneously. This could result in decreased response times if there are idle resources. It will however only result in increased overhead if the replicas are under heavy load. If 4 replicas are receiving requests from 12 nodes it does not make a difference to their load if each replica receives 3 full tasks or 12 smaller (1/4) tasks, the amount of work is still the same.

It is possible to use replication to avoid hotspots and reduce response times at the same time. The BitTorrent-protocol [37] lets downloading nodes download fragments from each other and thus relieving the original source of much of the load while decreasing response times. A similar approach can be used in a database system if the query processor is based on data shipping. However, it will require an extra effort to keep the replicas in a consistent state.

Replication can also increase the availability of a resource, but it is necessary to have a strategy in case the owner of the resource never returns to the system. Two possible strategies is to keep the replicas alive as long as they have a certain popularity level another is to give them a time to live. Both these strategies will only ensure extended, and not permanent, availability.

The use of replication also has its disadvantages. Without careful consideration, replication can be devastating with regard to the resource consumption. If the workload is mainly read operations, having multiple copies of an item is good for the performance [2]. On the other hand, if the core database is constantly being updated, then maintaining replicas requires a considerable amount of resources.

Replica Maintenance

When there are replicas in a system, some scheme must be defined in order to keep the replicas up to date. There are a variety of alternatives to choose from. The replica control protocols can be divided in two categories; strict and lazy replication [19, 2].

Strict means enforcing equivalence among database copies. In short, every copy must be consistent with each other after an update. The Read One Write All (ROWA) protocol is one such protocol. It transforms a read operation to a read from any of the available copies. Write operations are converted to a write to all the copies, thus ensuring consistency. Several optimizations on ROWA have been proposed. Protocols built on the concept of voting have also been explored. Basically, some form of voting process between the nodes in a distributed database is used to decide when to update.

Lazy replica control protocols allow updates without necessarily waiting for all replicas to be written to. An update is written to some of the copies, and propagated to the remaining replicas at a later time. This can lead the database into a state of inconsistency, which must be detected and resolved in some fashion.

Replication Strategy

A replication strategy governs when to replicate and where replicas are stored. The strategies can be divided into two categories, static and dynamic.

Static replication determines the replication parameters at connection time. This means that the number of replicas, what to replicate and where to send the replicas are determined when a site is added to the distributed database. The CAN [38] and Pastry [16] DHTs suggest this type of replication strategy to ease the transition when a node goes down. The number of replicas is usually small. There are several options for selecting the site to ship a replica to. It can be random, by sampling a subset of the network or in a predetermined manner according to the structure of the DHT. These can be supplemented with demand-driven replication, where any node can request a replica. This can be advantageous if the node will make frequent use of the object. The advantage of static replication is that it does not need any elaborate scheme to decide when to replicate, which makes the overhead non-existent. However, static replication does not take into consideration the shifting popularity of data. If a given set of data becomes popular, nothing is done to adjust to the increase in demand other than hoping the number of replicas is enough. In addition, resources are wasted by replicating data that are seldom or never accessed.

Dynamic replication strategies can be categorized in two groups [19]. The first group are those that aim to reduce communication cost by increasing locality. This is done by placing replicas at sites near where they are anticipated to be used. The second group measures the popularity of data, and adapts to changes in demand by replicating hot items. Examples of this strategy is the LAR scheme [39] and Beehive [36].

The two groups are not mutually exclusive, as it is perfectly possible to have both in the same system. Both are useful in a P2P database setting, though care must be taken to handle the added complexity. Resources must be used to maintain the replicas, and a mechanism is needed to avoid duplicate results from replicas.

To give an example of dynamic replication, the concept applied in the Beehive system will be presented. The purpose of Beehive is to reduce the lookup latency in a DHT by spreading DHT record replicas when they are needed. This idea can be adapted to a P2P database to improve query performance. Instead of replicating DHT records, parts of the database can be replicated and spread in lieu with increasing demand.

Beehive exploits the structure of DHTs that rely on prefix-routing. Prefix-routing implies that the identifier space is circular. The idea is presented in section 2.3.2. Every node is given an identifier in the circular identifier space. The objects inserted to the DHT are assigned identifiers, and are stored at the node with the most equal identifier. When routing requests for data, the request is routed to nodes with successively matching prefixes. The Beehive concept is that the number of hops to the destination can be reduced by one if a replica is inserted at every node preceding the destination by one. Preceding by one means there is a difference in one of the prefixes, and thus the destination node is in their routing table. The number of hops can be reduced even further by expanding the replication to the previously preceding points, and so on until the replica has been spread to all nodes. Figure 3.7[8] shows how the path to the destination can be made shorter by adding replicas. In the figure, the last routing hop comes from a level 2 node. If replicas were spread to level 2 nodes, this hop would be avoided. The same goes for the hop from level 1 to level 2, if replicas were spread to level 1.

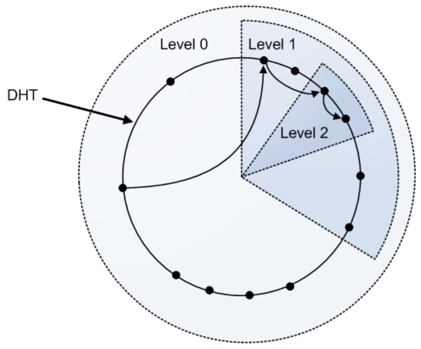


Figure 3.7: Replication levels and the effect on routing.

In order to use popularity to govern replication, a mechanism has to be in place to measure the popularity of an object. This is a function of the number of times an object has been accessed over a certain period of time, and is measured locally at each node. When an object is loses its popularity, it should no longer be replicated. In the Beehive model, the popularity of an object is measured as the aggregate of all accesses made to an object and its replicas. The aggregate value is gathered periodically, and is collected by having local values flow from the nodes at the lowest level of replication to the source copy. The source aggregates the values, and sends the final aggregate toward the lowest level. This way, all nodes keeping a replica knows when the data is no longer popular.

In the face of flash crowds, where certain objects become very popular in a short amount of time, this can easily flood the popular nodes. To decrease this problem, new requests for data can be delayed until the replicas are in place.

In a distributed system where the resources are not owned by the same organization, the willingness to participate in replication schemes may not be in everyone's interest. Even if the willingness exists, it may be that nodes lack the resources to store copies for others. There are schemes that create an incentive to participate in sharing, such as the Mariposa system [22]. This is not the focus of this master thesis, and will not be pursued any further.

Replica Granularity

The size of replicas is an important aspect of replication. Alternative granularities for replicas are entire databases, tables, parts of tables and tuples. Each granularity has its benefits and disadvantages:

[8]Figure adapted from [36].

- **Replicating databases**
  Measuring when to replicate when replicating the entire database, is easier than when the granularity is smaller. It is merely a matter of controlling the number of external accesses. There is less overhead since there are no partitions.

  On the other hand, replicating an entire database can be costly. If a replica leaves the network, replacing that replica means sending the entire database. Another problem is the popularity of the data being replicated. Most likely, only a fraction of the entire database will be popular data worth replicating. Shipping and maintaining unnecessarily large amounts of data is not optimal.

  Keeping in line with shifting hot spots is also difficult when replicating entire databases. Due to the potentially large amounts of data in a database, the time between the detection of a hot spot and the time the replica is in place might be longer than the lifetime of a hot spot.

- **Replicating tables**
  Using tables as the unit of replication makes it possible to discern popularity at a smaller granularity. This avoids replicating tables which are not a part of the hot spot. A disadvantage is an increase in overhead for building and maintaining a bigger index. Its size is governed by the number of popular tables at each node. The overhead can be decreased by increasing the level which decides when a table is deemed popular. The downside is that the time before replication begins, and thus relieving the hot spot, will be longer.

  There is a chance that not all tuples in the replicated table are popular. In the extreme case, only a single tuple is popular, and this would imply that an unnecessary amount of data was replicated.

- **Replicating parts of tables**
  There are several different approaches to partitioning tables, a survey of which is presented in the context of parallel databases [40].

  The simplest one, round-robin partitioning, maps the i'th tuple to site $i \bmod n$, where n is the number of sites in the partitioning set. Hash partitioning places tuples to a set of sites based on a hash function. In both these cases, the granularity is at the tuple level. This gives a high maintenance cost in a P2P setting.

  Range partitioning splits a relation into two or more ranges. These could be replicated at different nodes, which would enable parallelism while minimizing the size of the replicas. An optimizer would have to be in place to determine the best number of ranges. Having too many replicas would require the node storing the source copy to maintain an expensive amount of network connections. If the source copy is updated frequently, it might be a good idea to reassess the range partitioning. Shifting the contents of the replicas is potentially a costly operation. On the other hand, a smaller granularity gives a more detailed control over hotspots.

- **Replicating tuples**
  Replicating at the tuple level represents the extreme end of the spectrum.

It gives the most powerful control over replicas at the expense of the highest maintenance cost. Control means being able to exactly define hot spots and thus replicate only popular tuples. The cost of maintenance would be high because this alternative requires an index record for each replicated tuple.

The choice of replication strategy depends on the application, though an optimum solution would perhaps be to create a hybrid version that could adapt to the current environment. Creating such a hybrid would be a complex task, and will not be pursued here.

In the light of the PORDaS P2P environment, replicating at the table level is what makes the most sense. It incurs a medium impact on the index and the bandwidth consumption compared to the other alternatives. Additionally, it is easily incorporated into the current platform due to the design of the index.

3.3.2 Range Queries

If multiple sites in a distributed database can store data using the same schema, thus creating a distributed relation, it is possible to increase the performance of range queries. By knowing where ranges are stored, queries can be directed to those sites only. This is especially useful when many sites store data in the same relation.

One way of supporting range queries in a P2P network is to use a centralized index. The index would store global knowledge of the values in the network so that it could relay range queries. This approach suffers from a single point of failure and poor scalability. A completely decentralized way of supporting range queries is to flood the network with requests, as it is done in the Gnutella network. The benefits are that there is no single point of failure and each node only keeps track of its local information. But flooding the network with queries takes a high toll on performance.

When using a DHT to structure a P2P network, there is no direct support for range queries. A range query has to be resolved by asking every site that stores the relation.

There have been several proposals for handling range queries in structured P2P networks, either by using an alternative overlay structure or building an extra layer on top of a DHT. The alternatives are presented in the following section:

- **Alternative Overlay Structure**
  BATON [13] is a binary balanced tree overlay structure that is not based on hashing. The motivation for using a tree structure is because it is a heavily used data structure in traditional databases. For fault tolerance BATON employs horizontal links between the nodes in each level of the tree. BATON guarantees a cost for both exact and range queries to be performed in $O(\log N)$ steps in a network of N nodes.

- **Range hashing**
  One approach [41] used locality sensitive hashing to place ranges of values at the same node with high probability. Intuitively, this is in conflict with

attaining a uniform data distribution. Additionally, the range hashing is approximate, which means the probability of finding correct answers is less than optimal.

- **Range index**
  Information about ranges and where they are stored could be put in an additional index in the DHT. This way, queries can be constrained to only those sites that have data in the requested ranges. The cost is to maintain an extra index.

  P-Tree [42] is based on the B+-structure and has a Chord [43] implementation as its overlay network. The idea is building a B+ index on top of Chord that holds range information. In a sense it has similarities to the R-Tree [44] for spatial databases. The P-tree gives a theoretical guarantee of $\log N$ for both exact and range queries.

It would be interesting to try these the alternatives in PORDaS, but in order to constrain the scope of the master thesis, they will not be pursued any further here.

3.4 Testing

This section describes different aspects related to testing that must be taken into consideration when designing PORDaS. One must take into account what the tests should include, which resources that are available and the project's timeframe.

The intention behind PORDaS is to create a scalable database system spread over a wide area and possibly heterogeneous networks. If experiments should support such a claim one would have to run the experiments in a similar environment, 100.000 nodes and above [45]. This is obviously an unfeasible task without running multiple node instances per computer and simulate network communication between them.

Simulating multiple nodes will require modifications to PORDaS. The network layer must simulate delays between local nodes and the nodes must share their access to the processor, storage and network with other nodes. Creating a proper simulation environment is just a too complex task for this project's timeframe.

The computers available for testing will probably be a cluster with $20-60$ nodes so the only realistic goal is to verify that each node behaves as expected and to identify problem areas and bottlenecks. This can be done by running different scenarios and measuring the performance in each scenario.

Ideally, one should be able to measure the time spent on every single task. However, this will affect the performance of PORDaS and generate a quite large amount of data. The amount of data can be reduced by only storing aggregates like average, minimum and maximum values.

Given the resources and timeframe of this project a realistic goal would be to verify that the system behaves as expected on the available hardware and to identify possible weaknesses. The tests should not be concerned with subjects that are not in the scope of this thesis such as robustness and problems related to churn.

Chapter 4

PORDaS

This chapter presents the finished PORDaS. The first section gives and overview of PORDaS and its architecture. The following sections presents the main components while the final section presents two applications that are built on PORDaS.

4.1 Overview

The new version of PORDaS is a distributed database with support for parallel execution of queries. It is built as a database layer on which larger application can be built. The main addition to PORDaS is the improved distributed query processor. In the previous version, queries could only target tables from a single site, and they were not parsed but sent directly to the database layer. In order to accommodate joins between sites and to free the design from specific database implementations, a new query language was created. A new parser had to be built from scratch in order to support the new features. The rest of the query processor had to be built as well, and a lot of effort went into the planner and executor. It can now build both bushy and linear query trees, and the execution can either be centralized or distributed. These are explained in the section on query processing. Another big improvement is the introduction of a pipelined process, which is a much better match for a P2P database. Previously, results were sent in one single message, which delays the result until the entire query is processed. Much of the internal workings of PORDaS have also been improved. Message sending in the DHT has been made more efficient, and results are no longer routed but sent directly using sockets between source and destination.

Some of the desired functionality of a P2P database presented in the analysis has not been implemented. This includes introducing replication, creating a range index, having more ways to perform joins and adding a dynamic executor. Analyzing the field to find sensible solutions was preferred rather than focusing on the implementation. Also, developing a distributed database proved more difficult than first expected, regardless of the simplifications. This is especially true for the planner and optimizer, which turned out to be very complex subjects.

Architecture

This section present the architecture of PORDaS, along with a brief explanation of the main components. Using the previous version of PORDaS as a foundation required some remodelling, but the main components are the same. Figure 4.1 shows how PORDaS is structured.

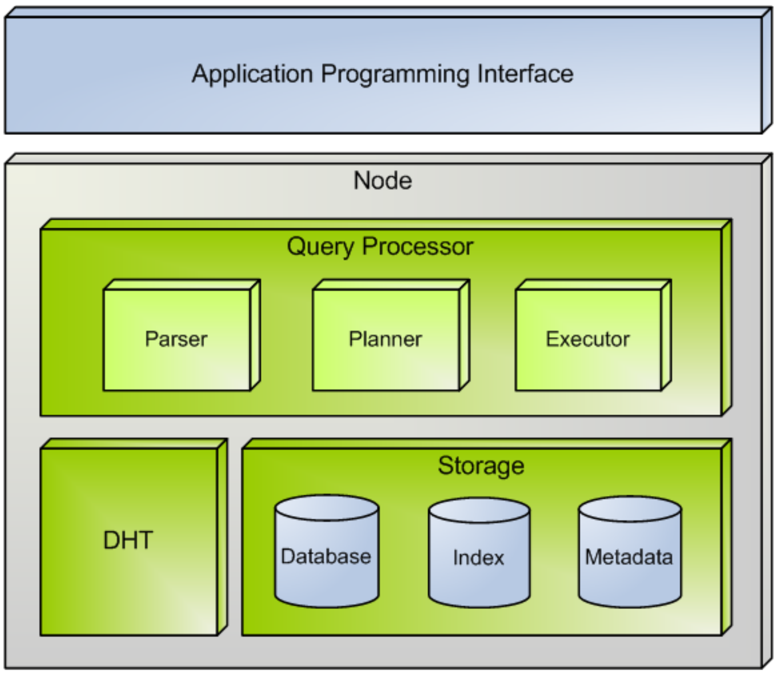


Figure 4.1: The PORDaS architecture

The node is the main component, encapsulating all the other components in PORDaS. It binds them together and provides an entry point to the outside through an Application Programming Interface (API). The API enables the building of new applications on top of PORDaS while protecting the inside of the node. The application layer can interact both with the local database and query data at other sites transparently, without users having to know where the data is located.

The nodes in a PORDaS network are loosely connected through a DHT (section 2.3.2). Its purpose is to enable resource location and to route requests for data. It also enables message sending and reception. When a message is received, it makes sure the message is unwrapped and that it reaches the appropriate component. The communication section explains the DHT in depth.

The storage component keeps track of all the data stored at a node. It holds the local database and a metadata repository that manages information about local tables. The storage component also stores parts of the distributed index, which all the nodes in the system participates in sharing. The index holds information about the contents and location of other tables in the system.

The query processor consists of a parser, a planner and an executor. It takes queries as input from the application layer, creates a plan and executes it. The resulting tuples are pipelined back to the application layer.

4.2 Communication

This section explains how nodes communicate in PORDaS.

Except when returning query results, every message in PORDaS is routed through the DHT layer. Messages are sent for keyword and query requests, to resolve sub queries and to maintain the distributed index. Appendix C gives an overview of the messages in PORDaS. When returning results, direct TCP connections between the nodes are used for improved performance.

The DHT layer in PORDaS is created as an interface. This means that any DHT can be used for message routing, as long as it complies with the DHT interface. The interface specifies the required functions for joining and leaving a network, and for routing messages based on an identifier.

PORDaS currently uses the FreePastry DHT [15]. It was chosen because it was easily integrated, and because of its platform independence. PORDaS will in theory run on any system that supports Java, which is virtually everything. It has been tested on both Windows and Unix systems, without any problems. However, Pastry is not the most efficient DHT in terms of routing speed and overhead. According to its authors, the Bamboo DHT [46] is better in that respect, though it is lacking in documentation and platform independence.

4.3 Storage

This section presents the elements of the storage component. These are the local database, the index and the metadata repository.

Local database

PORDaS operates internally on an object relational data model. It only supports abstract data types, but could be extended to include more object relational functions, such as inheritance, methods and collections. The choice of a object relational model was due to it being the current industry standard, and to prove that it could be used in PORDaS.

A PORDaS node needs access to a database in order to store data. In the spirit of modularity the database is only defined as an interface. This allows PORDaS to, in theory, use any kind of storage. The only requirement is that a database connector that adheres to the database interface is created. PORDaS is not concerned with how the connector maps the object relational model to the underlying database's storage model. Figure 4.2 illustrates the concept.

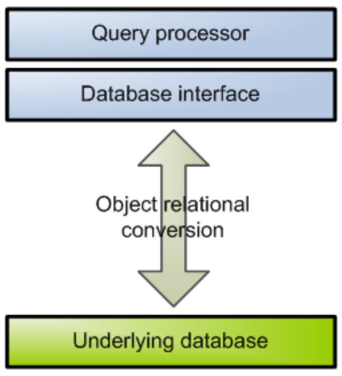


Figure 4.2: Database interface

There is currently only a connector for Derby which is an embedded database. This means the database can be packaged along with the application and thus avoiding an installation procedure.

Index

The distributed index serves two purposes; searching for schema definitions using keywords and finding the locations of arbitrary tables. Section 2.4 explains the rationale for storing keywords in the index. The ability to locate tables is needed by the distributed query processor.

When a new table is inserted into the system, several hash values are computed; one for each keyword associated with the table and one based on the schema definition. Figure 4.3 shows the content of the index as it is stored at each node. For each hashed keyword record in the index, there is a list of table definitions with a respective description. It's a list because a keyword can be used to describe more than one table. The same goes for the table index. More than one node can store the same table, which means that queries targeting a certain table get the concatenated result from all nodes storing that table.

Each node in a PORDaS network are responsible for separate parts of the index, based on the node identifier used in the DHT. In order for this to work, the index records are assigned identifiers too. These are computed by hashing the value to be indexed, which is either a keyword or a schema definition. A record is then mapped to a node by storing it at the node with the largest matching identifier. This node is found by the DHT routing mechanism. To retrieve this record from the index, the record identifier is used as the key in a DHT lookup. Figure 4.4 gives an example of a set of nodes spread in a simplified version of the circular identifier space used in FreePastry. Next to each node are the keyword and table records it stores in the distributed index. For example, node 8 is responsible for all records from 5 to 8. The figure also illustrates how a lookup for key 4 is resolved when originated from node 8.

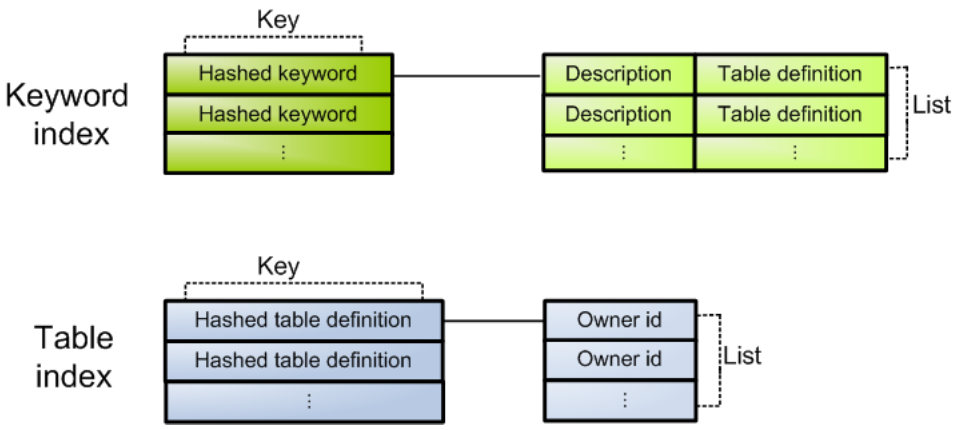


Figure 4.3: The keyword and table index

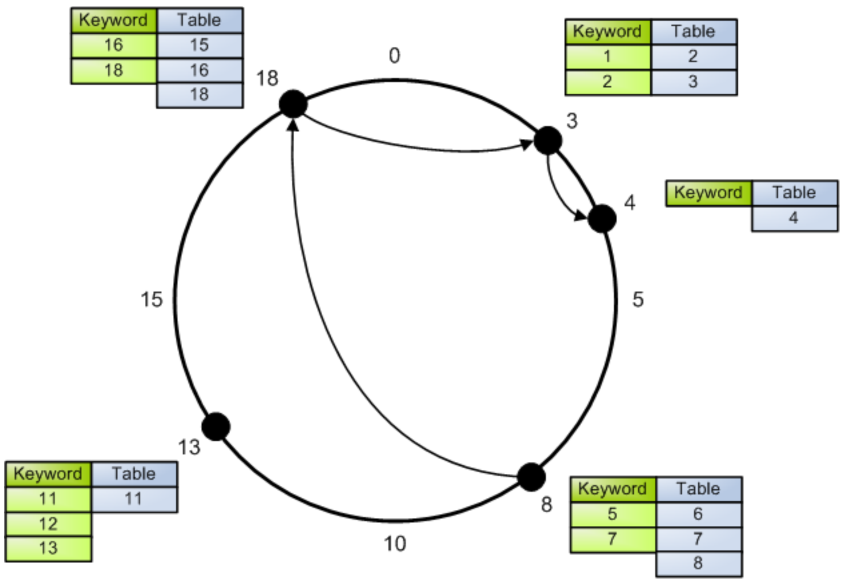


Figure 4.4: The keyword and table index

Figure 4.5 gives an example on how the index can be distributed among 36 nodes. The data was collected when testing PORDaS. As can be seen in the figure, some nodes are responsible for larger parts of the index than others.

The index makes PORDaS location transparent. This means that in order to query any table, local or external, all that is required is the schema definition. By hashing the schema definition, every table that has this schema is found.

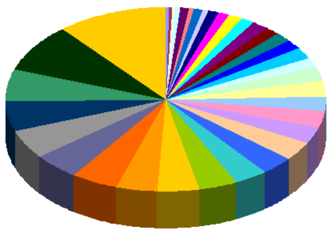


Figure 4.5: Distribution of index entries

Metadata

A node needs knowledge about the contents of its database, which is referred to as metadata. It is needed when entering the index, when keeping the index up to date, and when the parser is checking query references.

Metadata is a list of table objects that encapsulate information about every table in the local database. An object contains the name of the table and a list of the table's attributes. For convenience, the hashed value of the table's schema is also stored. Storing the hashed value avoids having to rehash the schema every time it is needed. In addition, every object in the metadata list has a set of keywords and a textual description. Figure 4.6 summarizes the contents.

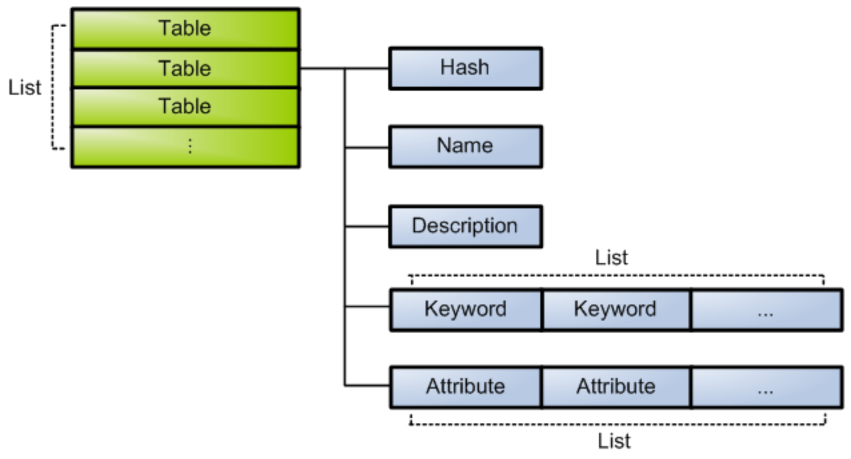


Figure 4.6: The contents of metadata

The metadata is frequently accessed, so it is an advantage to have a representation in memory to increase performance. The alternative would be to extract metadata from the database each time it is needed, resulting in more disk accesses. The metadata is therefore loaded into memory at startup, and kept there as long as the application is running.

Whenever a new table definition is discovered, either when searching for keywords or when querying using hashed identifiers, the table definition is added to the metastore. By caching table definitions this way, time is saved the next time this table is accessed. Additionally, by storing external table definitions locally, they can be queried using their table name as the identifier instead of using the hashed schema value[1].

Resource identifiers

Every resource in PORDaS needs a unique identifier. Each node has a 40 bit identifier, and so has each schema and keyword. Resource identifiers are unique

[1]If there are more than one table with the same name in the metastore, they can't be distinguished by name. In this case, the hashed schema value must be used instead.

in the sense that identical resources generates identical keys[2]. The keys are generated by applying a cryptographic hash function. Keyword identifiers are created by simply hashing the keyword, while schema identifiers are created by hashing the entire schema-definition and thus if a schema is changed it is regarded as a completely different table.

Even though it was never realized PORDaS is designed with replication in mind. Replicas would be read-only. In addition to replicas, it is also possible to create the exact same schema on multiple nodes. This raises the need to identify schema instances. A schema instance identifier is an 80-bit identifier that is a concatenation between the schema identifier and the master node's node identifier. This causes all schema instances to be indexed at the same node and index entries with the same instance identifier are replicas.

Node identifiers are randomly generated, but are assumed to be unique due to the large identifier space ($2^{40} \approx 10^{12}$). Another approach would be to give an independent source responsibility for the designation of identifiers. Coupled with security certificates it would add authentication to PORDaS.

Maintenance

Soft state (see section 2.4) is used to maintain the distributed index. It means that every index record has an associated Time-To-Live (TTL) value. When it expires, the record is deleted from the index. Thus, in order for a node to keep its index records in the system, they must be continually refreshed. After a node leaves the system, either voluntarily or because of an error, its index records will perish when the TTL reaches zero. This is fits nicely with the P2P environment, where periods of fluctuations must be handled. If a node goes down, the index records it holds for others disappears. Due to soft state, all nodes that had some of their records at the failed node, will eventually discover the failure. Thus, they will adapt to the failure by resubmitting their missing records to the DHT.

4.4 Query Processing

The fundamentals of distributed query processing is presented in the background section 2.1 and in section 3.2 in the analysis chapter. This section gives an overview of how the theory was adapted to create a distributed query processor in a P2P environment.

First, an overview of the processor is given, and then the query language is explained. The last part walks through the process step by step.

[2]The identifiers can only be regarded as pseudo-unique since there is a small chance that two different resources could generate the same key.

Overview

Figure 4.7 shows how queries are resolved in PORDaS. The structure is the same as for traditional distributed systems, though the observant reader will notice the lack of an optimization step. This is left as future work as there was not time to implement it.

The processor can handle multiple queries at a time, through the use of threads and queues of messages and results. The number of threads was a significant challenge, both in terms of performance and being a big source of errors. In retrospect, a single threaded design for the query processor would be better. Operators would then be implemented as iterators and not as threads.

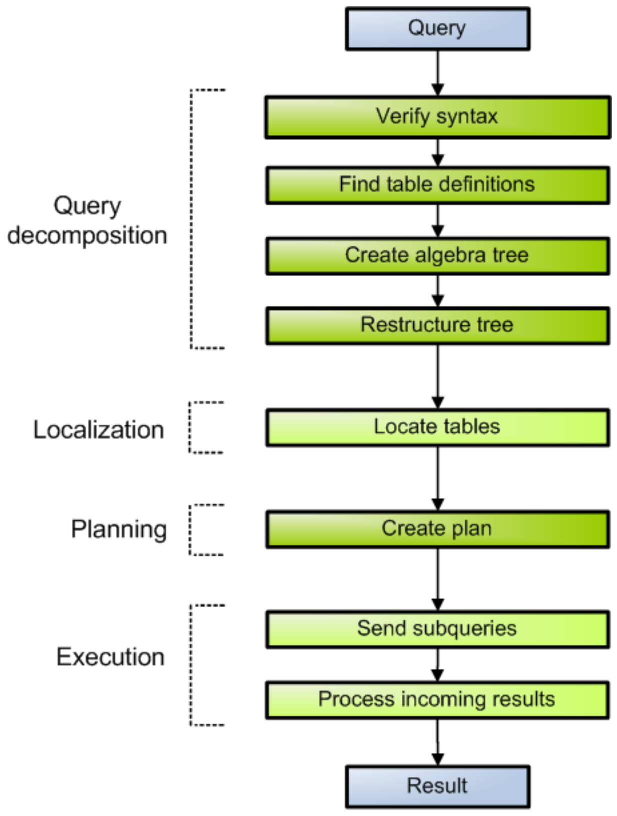


Figure 4.7: The life of a query

Query Language

The query language is a simplified form of SQL. The supported functionality is selection using less than, equals or larger than, projection, cartesian products and joins between tables. More than one conditional can be added using the AND keyword. To reduce the complexity, negation and combination using OR was not implemented.

Insertion is also supported. A complete definition in Backus-Naur form can be found in appendix B. The following shows the basic structure of an insertion:

INSERT INTO TableReference VALUES(Attribute1, Attribute2 , ..., AttributeN)

A selection has the this syntax:

SELECT Table1.Attribute1, ..., TableN.AttributeX
FROM Table1, ..., TableN
WHERE Table4.Attribute1 < 123 AND Table2.Attribute1 = Table4.Attribute2

Tables in a query can be referenced by their name or by their hashed identifier. To distinguish between the two, a ♯ is used as a prefix when referencing hashed identifiers.

Query Decomposition

Decomposing queries from a textual representation into an algebra tree is done by the parser component. The PORDaS parser is kept simple, as little effort is used on informative error-handling or reforming inefficient queries. There will be no attempt to remove redundancy nor checking for semantic correctness[3] as creating a user friendly interface is not the main goal of the master thesis.

When a new query is submitted, the parser's first task is verifying the query syntax. The syntax is checked against the Backus-Naur form defined in appendix B. The next task is to verify type correctness, which is making sure the relations and attributes referenced in the query actually exist. The operations in the query are checked against the type of each attribute as well, making sure they match. Before this can be done, the table definition for each table referenced in the query must be fetched. First, the local metastore is searched. If one or more table definitions are lacking, they must be fetched from the distributed index. This delays the query until all tables are accounted for.

The next step is creating the initial algebra tree. The final tree is either bushy or linear, depending on the settings. The first part of the tree is always the same for both types. First, the projections are defined as the root, followed by each selection in the query. In the case of building a linear tree[4], any cartesian products are added as the left child of its parent. If there are joins in the query, these are added as the last part of the tree, in the same fashion as cartesian products. When building a bushy tree, the goal is to balance the tree as much as possible, catering for parallel execution. This is done by maximizing the number of cartesian products and joins with two operators as children.

Restructuring the tree is done to get a better tree. The projections and selections in the algebra tree are pushed as far down as they can be, and in doing so, the size of intermediate results will be reduced during execution.

[3]A query is semantically correct if all of its elements contribute to the final result.

[4]PORDaS builds left-deep trees, which are a subset of linear trees. A left-deep tree is always extended along the leftmost path of each operator, with only base relations as the right child of the operators.

Localization

Localization means finding every site that has a table referenced in the query. These are found by querying the distributed index, and are sorted according to the table that they hold. Using caching to improve the localization process is not done, because of the potentially volatile nature of the P2P network. Nodes that store one of the tables in the query might have left or joined the network since the last time they were queried.

Planning

When the locations of every table in the query are found, a plan for the execution can be devised. There is no optimization involved, as there is no way to retrieve statistics needed to make informed decisions using a cost function. Instead, the planner uses a simple heuristic to create plans. These plans are either centralized or distributed.

Centralized Plans

A centralized plan is a plan where the required base relations are fetched to the initiating site so that the operators in the query can be resolved locally. In the spirit of reducing the amount of network traffic, any selection and projection on base relations are executed at the remote sites before the streaming of results is started. Figure 4.8 gives an example of a centralized plan. The dotted lines indicate network communication. It is important to note that the data fetched from a base relation in the figure may include contacting one or more sites.

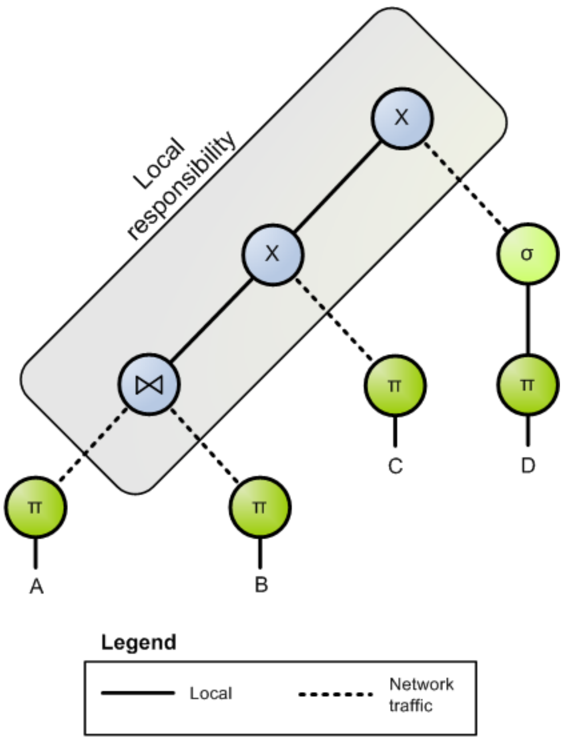


Figure 4.8: Centralized plan

Distributed Plans

In a distributed plan, the responsibility for executing the query tree is distributed among the set of nodes that store tables referenced in the query. If statistics about each table was available, like cardinality, maximum and minimum values, this information could be used to restructure the tree in a beneficial way. As this is not the case, predefined rules are used instead. The rule is best explained by an illustration, see figure 4.9. It gives an example of a conversion from an algebra tree to a distributed plan. The first rule is to always calculate cartesian products at the initiating node, which avoids sending too much data over the network. It is conceivable that in certain cases it might be better to distribute cartesian products as well, but it is assumed to not be the average case.

The second rule is to delegate the resolution of joins to one of the owners of the leftmost table in the join tree. The choice is arbitrary, it could have been any of the owners. The chosen owner will receive the entire join subtree. If this subtree has any more joins in it, these joins are delegated in the same manner as well. In the figure this can be seen as the second join under the cartesian product is delegated to two separate nodes.

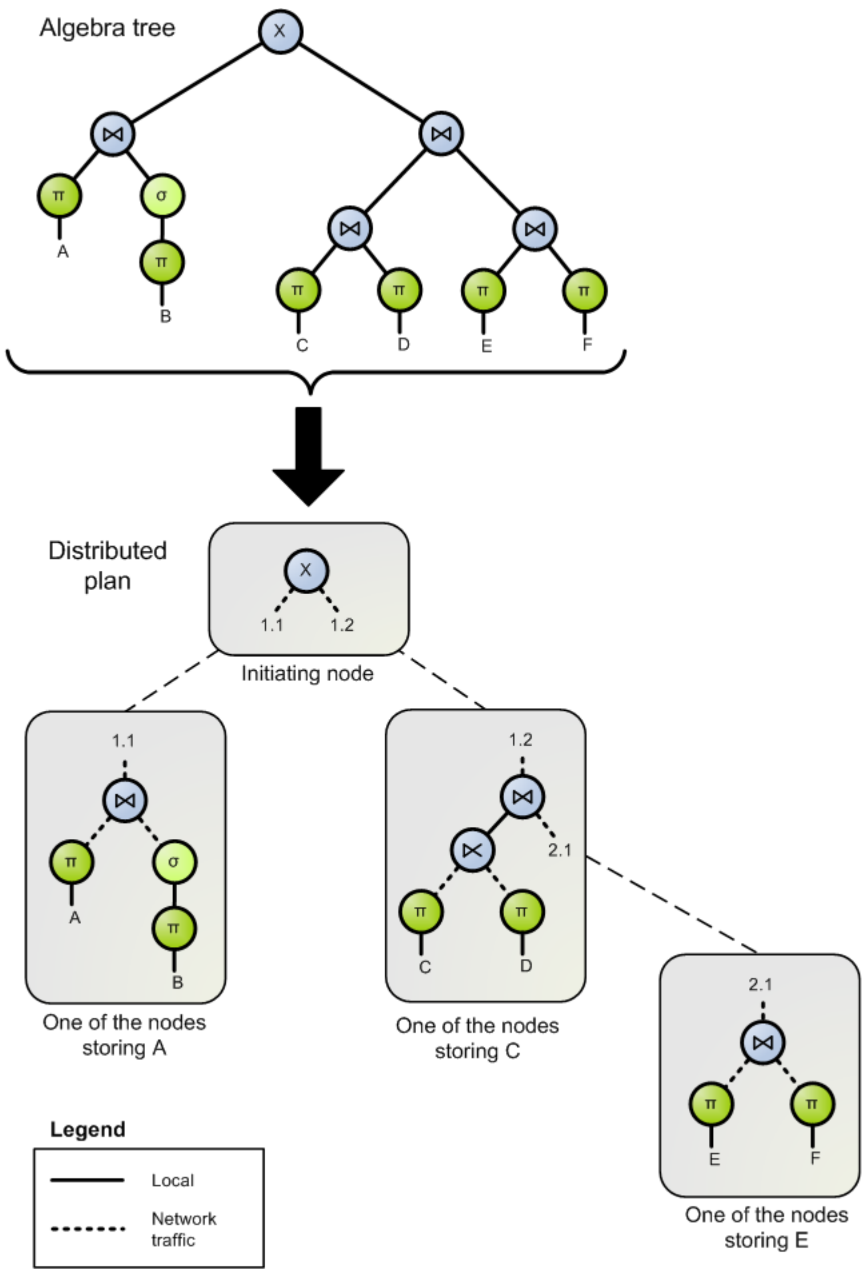


Figure 4.9: Conversion from an algebra tree to a distributed plan

Execution

The execution phase begins with the initiating node requesting data from all base relations in the query. If the query plan is a distributed query plan subtrees are delegated to other nodes. The execution is pipelined, which means that processed tuples are sent up the tree as soon as possible. This avoids having to store temporary caches with intermediate results. To know where a tuple belongs at the receiving end, all tuples sent across the network are tagged with an identifier. The identifier uniquely specifies the query and the point in the query tree where the tuple is expected. This can seen in the plan in figure 4.9. For instance, the children of the cartesian product are tagged as 1.1 and 1.2. At the sites responsible for those parts of the query, every resulting tuple is tagged with 1.1 or 1.2, respectively.

Joins

The query processor relies on a naive join algorithm based on a simple nested-loop join which makes dataflow query processing possible (Figure 4.10). The key difference is that it does not have an inner or outer table. Tuples from one operand are joined with all previously-seen tuples of the other operand. The join algorithm changes to a regular nested-loop join when one of the operands has reached 'end of stream'. When the join operator has processed both input streams it will send an 'end of stream' message. Figure 4.10 shows an overview of the join operator and an example of how the join operator runs through the cross-product space when operand B arrives faster than operand A.

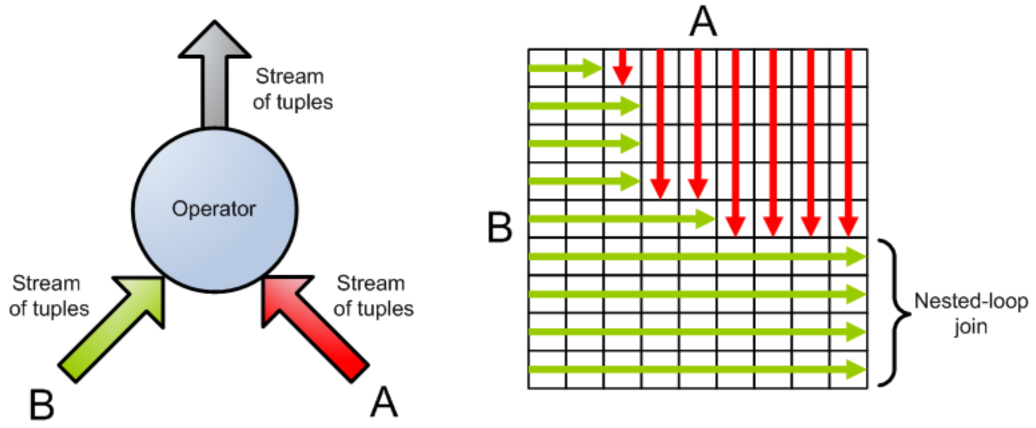


Figure 4.10: Pipelined join operator

The algorithm does not handle large data volumes, it assumes that the entire operation can be done in main memory. An additional drawback is that it is inferior to algorithms like pipelining hash join (Chapter 3.2.1) when it comes to equi-joins. It is however more versatile since it can perform any kind of join. A database system should ideally

Failed Queries

A query can fail if any of the nodes it relies on goes down. Section 3.2 in the analysis chapter presents different strategies for coping with failure. The simplest strategy is chosen for PORDaS, where a failed query times out and returns a failure message.

4.5 Example Applications

This section presents two of the applications that were built using PORDaS as a distributed database layer. They were created for testing and demonstration purposes. A third program that simulates user behavior is presented in chapter 5 on testing.

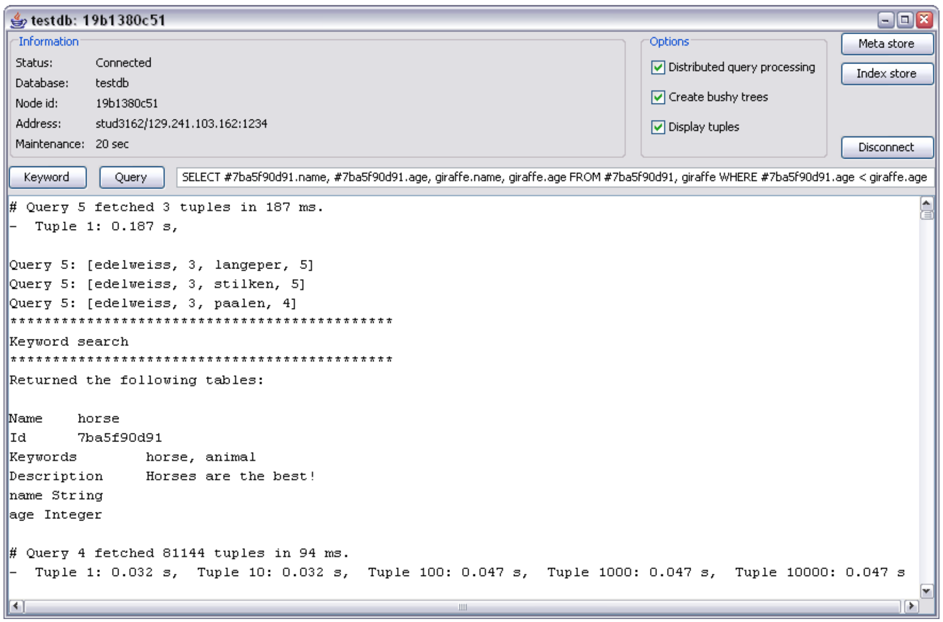
Graphical User Interface

The first application that was created enables access to a distributed database through a graphical user interface (GUI). A screenshot can be seen in figure 4.11. The application can both create a new PORDaS network, and it can connect to existing networks by using the address of an already connected node. When connected, interesting schema definitions can be found by performing keyword searches. Queries targeting these schemas can then be formed, and sent as requests to the PORDaS layer. Here, the queries are executed, and answers are pipelined back to the GUI, which displays them in the output area as soon as they arrive[5]. Options can be set such that queries are either resolved locally or divided as sub queries and executed in a distributed fashion (see section 4.4). The structure of the trees created by the planner can be set to be either bushy or linear (see section 3.2).

The GUI offers two views of the internal state of the database. The contents of the metastore and the index can be seen on request. These views were handy in the debugging process. A screenshot of the index is shown in figure 4.12. It displays the keyword and table records that are stored at this node. As can be seen by the selected keyword in the keywords list, there are 4 schemas in the system with that keyword[6]. The selected record in the table owner index shows there is only one node in the network that has tuples in this table.

[5]Showing tuples as they arrive is optional. Refreshing the GUI affects the performance of queries, especially when they are large.

[6]The “animal” keyword



testdb: 19b1380c51

Information

Status: Connected
Database: testdb
Node id: 19b1380c51
Address: stud3162/129.241.103.162:1234
Maintenance: 20 sec

Options

[✓] Distributed query processing
[✓] Create bushy trees
[✓] Display tuples

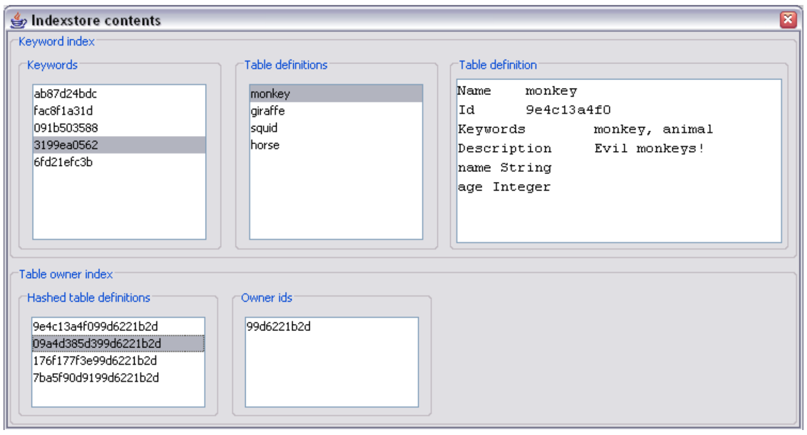
```
SELECT #7ba5f90d91.name, #7ba5f90d91.age, giraffe.name, giraffe.age FROM #7ba5f90d91, giraffe WHERE #7ba5f90d91.age < giraffe.age
```

```
# Query 5 fetched 3 tuples in 187 ms.
-  Tuple 1: 0.187 s,

Query 5: [edelweiss, 3, langeper, 5]
Query 5: [edelweiss, 3, stilken, 5]
Query 5: [edelweiss, 3, paalen, 4]
*********************************************
Keyword search
*********************************************
Returned the following tables:

Name    horse
Id      7ba5f90d91
Keywords        horse, animal
Description     Horses are the best!
name String
age Integer

# Query 4 fetched 81144 tuples in 94 ms.
-  Tuple 1: 0.032 s,  Tuple 10: 0.032 s,  Tuple 100: 0.047 s,  Tuple 1000: 0.047 s,  Tuple 10000: 0.047 s
```

Figure 4.11: GUI application

Textual interface

The textual interface is yet another presentation layer built over PORDaS. It has basically the same capabilities as the GUI application, and was created for use on platforms without a graphical interface. This was handy when testing the application through a UNIX shell. Figure 4.13 illustrates its use.



```
ab87d24bdc
fac8f1a31d
091b503588
3199ea0562
6fd21efc3b
```

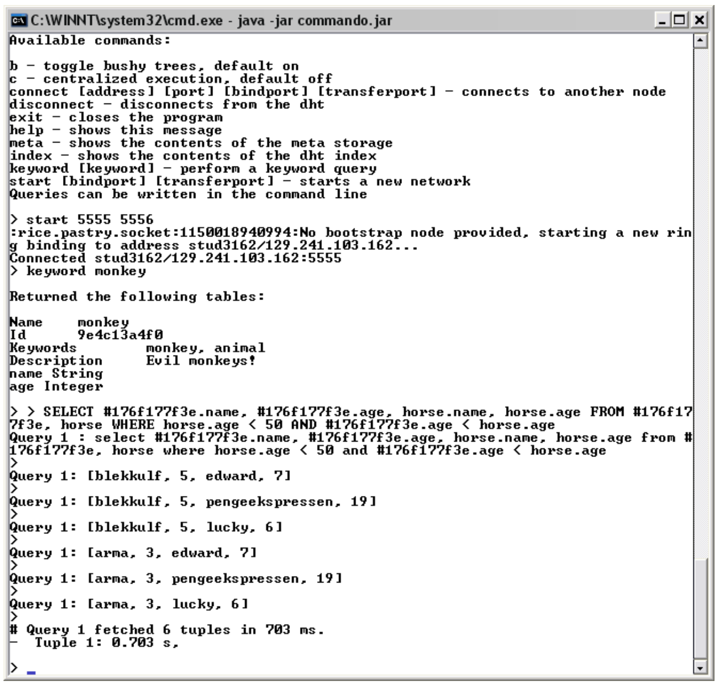
```
monkey
giraffe
squid
horse
```

```
Name     monkey
Id       9e4c13a4f0
Keywords         monkey, animal
Description      Evil monkeys!
name String
age Integer
```

```
9e4c13a4f099d6221b2d
09a4d385d399d6221b2d
176f177f3e99d6221b2d
7ba5f90d9199d6221b2d
```

```
99d6221b2d
```

Figure 4.12: Contents of the keyword and owner index



```
Available commands:

b - toggle bushy trees, default on
c - centralized execution, default off
connect [address] [port] [bindport] [transferport] - connects to another node
disconnect - disconnects from the dht
exit - closes the program
help - shows this message
meta - shows the contents of the meta storage
index - shows the contents of the dht index
keyword [keyword] - perform a keyword query
start [bindport] [transferport] - starts a new network
Queries can be written in the command line

> start 5555 5556
:rice.pastry.socket:1150018940994:No bootstrap node provided, starting a new rin
g binding to address stud3162/129.241.103.162...
Connected stud3162/129.241.103.162:5555
> keyword monkey

Returned the following tables:

Name     monkey
Id       9e4c13a4f0
Keywords         monkey, animal
Description      Evil monkeys!
name String
age Integer

> > SELECT #176f177f3e.name, #176f177f3e.age, horse.name, horse.age FROM #176f17
7f3e, horse WHERE horse.age < 50 AND #176f177f3e.age < horse.age
Query 1 : select #176f177f3e.name, #176f177f3e.age, horse.name, horse.age from #
176f177f3e, horse where horse.age < 50 and #176f177f3e.age < horse.age
>
Query 1: [blekkulf, 5, edward, 7]
>
Query 1: [blekkulf, 5, pengeekspressen, 19]
>
Query 1: [blekkulf, 5, lucky, 6]
>
Query 1: [arma, 3, edward, 7]
>
Query 1: [arma, 3, pengeekspressen, 19]
>
Query 1: [arma, 3, lucky, 6]
>
# Query 1 fetched 6 tuples in 703 ms.
-  Tuple 1: 0.703 s,

> _
```

Figure 4.13: Textual interface

Chapter 5

Tests and Results

This chapter presents how PORDaS was tested and the results of the testing. It consists of an overview of the test setup, the applications used for testing and each of the tests. Finally, the results are given in a graphical form which forms the basis of a discussion.

5.1 Test Setup

This section presents the test environment, the possible configurations and the type of measurements that were made.

Resources

The tests were conducted at one of the computer labs at NUST[1], hosting 80 computers with decent hardware[2] and Microsoft Windows XP. Due to technical difficulties[3] the number of available computers was reduced to 36.

Using a computer lab meant manually installing the PORDaS software at each computer which is a time consuming operation. The tests had to be carried out during nighttime to avoid disrupting the daily use of the lab. This was all a bit unfortunate since the program was not ready to run tests until late in the project. There was not room for many tests during the one night that could be used either.

Having 36 nodes in the DHT network means that there was at most one routing hop between any two nodes. The problem with using a cluster connected in a LAN is that the network is optimal compared to what would be the case for a real world PORDaS system.

[1] Norwegian University of Science and Technology

[2] 3 GHz Pentium 4, 1GB RAM

[3] The system administrator had difficulties with disabling the Windows Firewall. It had to be done manually on a smaller set of nodes.

There are also several UNIX clusters available at NUST that could have been used. These vary in size from seven nodes to over sixty, and consist of relatively powerful computers. Use of the clusters is streamlined through batch job queues, which significantly eases the task of deploying PORDaS. The seven node cluster was used initially while developing the test environment. The larger clusters were used by others as well. Using these clusters would have caused a lot of waiting since PORDaS need access to all nodes simultanously. Other applications would have been shut out during a simulation as well.

A more realistic testing environment can be found at PlanetLab. The PlanetLab project [47] has over 600[4] nodes in its planetary network. NUST has yet to enter due to the cost of joining[5]. It would have required a lot of effort to deploy and test PORDaS on PlanetLab as well, which rendered it unfeasible for this project even if NUST did have access.

Parameters

The test server could create different tests by changing these parameters:

- **Number of tables at each node**
  Having more tables in the local database means the node has to maintain more state in the DHT. This interprets into more traffic due to the maintenance mechanism.
- **Number of tuples in a table**
  The number of tuples in a table restricts the maximum amount of data that can be returned from a query.
- **Length of test**
  In order to make the collected data less susceptible to variances due to the randomness of the query process, the length of the test can be increased. In effect, the sample size is increased so that the standard error is decreased.
- **Number of nodes**
  If there were enough available computers, varying the number of nodes could indicate how PORDaS would scale to a larger network. Since there relatively few computers in the test environment, it has little purpose to alter this value. It will be kept at 36 for all the tests.
- **Request interval**
  The time between two consecutive requests defines how often a new query will be created. It can either be defined as a constant or be randomly distributed by a poisson distribution.
- **Number of concurrent queries**
  Setting the number of concurrent queries means halting when a certain number of queries are running. When the network is congested, setting this value low can avoid further worsening of traffic.

[4]677 nodes as of June 2006.

[5]PlanetLab requires that each participating institution donates at least two computers. The minimum hardware specification is 3.2 GHz Pentium4 or AMD 3200+ processor, 4GB RAM and 320 GB hard drive space.

- **Number of active nodes and sharing nodes**
  An active node periodically requests, directed at sharing nodes' data, with the predefined request interval. A sharing node is a node whose table definitions are distributed among the active nodes.
- **Type of Query Processor**
  Queries are resolved either in a centralized or a distributed manner. Section 4.4 in the PORDaS chapter gives an explanation.
- **Type of query trees**
  The trees produced in the planner can be either bushy or left deep. Section 3.2 illustrates their differences.
- **Result size**
  The size of the final result can be constrained to a maximum value. A standard deviation can be supplied to get different sized queries. By setting this value, very large and very small results can be avoided.
- **Number of tables in a query**
  The number of tables referenced in a query can be set to get a certain control over the number of nodes involved in a query. A standard deviation can be defined to get variation.
- **Number of joins in a query**
  Setting this number decides the number of joins a query.

Measurements

The tests measured:

- **Response time**
  The response time of a query is measured as the time from the query is sent until the last tuple is received. The maximum, minimum and average response times are recorded.
- **Throughput**
  Throughput is measured as the number of queries processed per second, and is calculated as the number of responses received divided by the length of the test.
- **Number of tuples received**
  The number of tuples received says something about the amount of data transferred through the network.
- **Index size**
  Knowing the index size for each of the nodes in the system enables the calculation of the distribution of index records.

5.2 Test application

The tests are performed by a test application which simulates the activity of a regular PORDaS node. The test application connects to a centralized server to ease the administration and collection of statistics (Figure 5.2). It is only the test application which uses the client-server model (figure 5.1), PORDaS is still P2P.

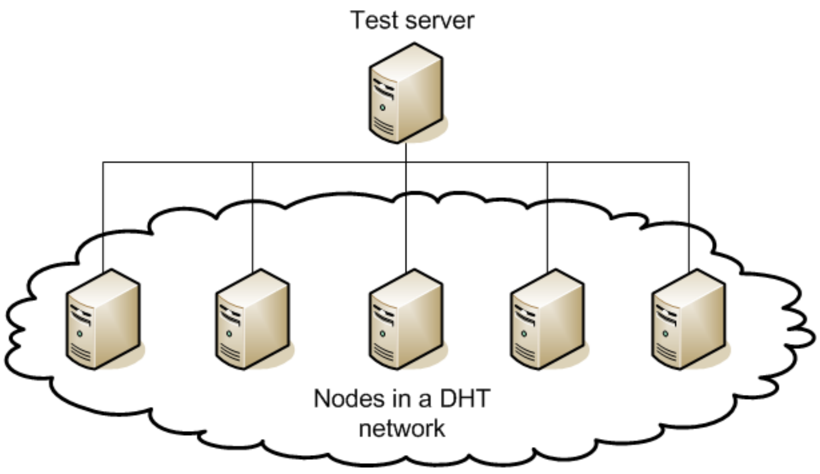
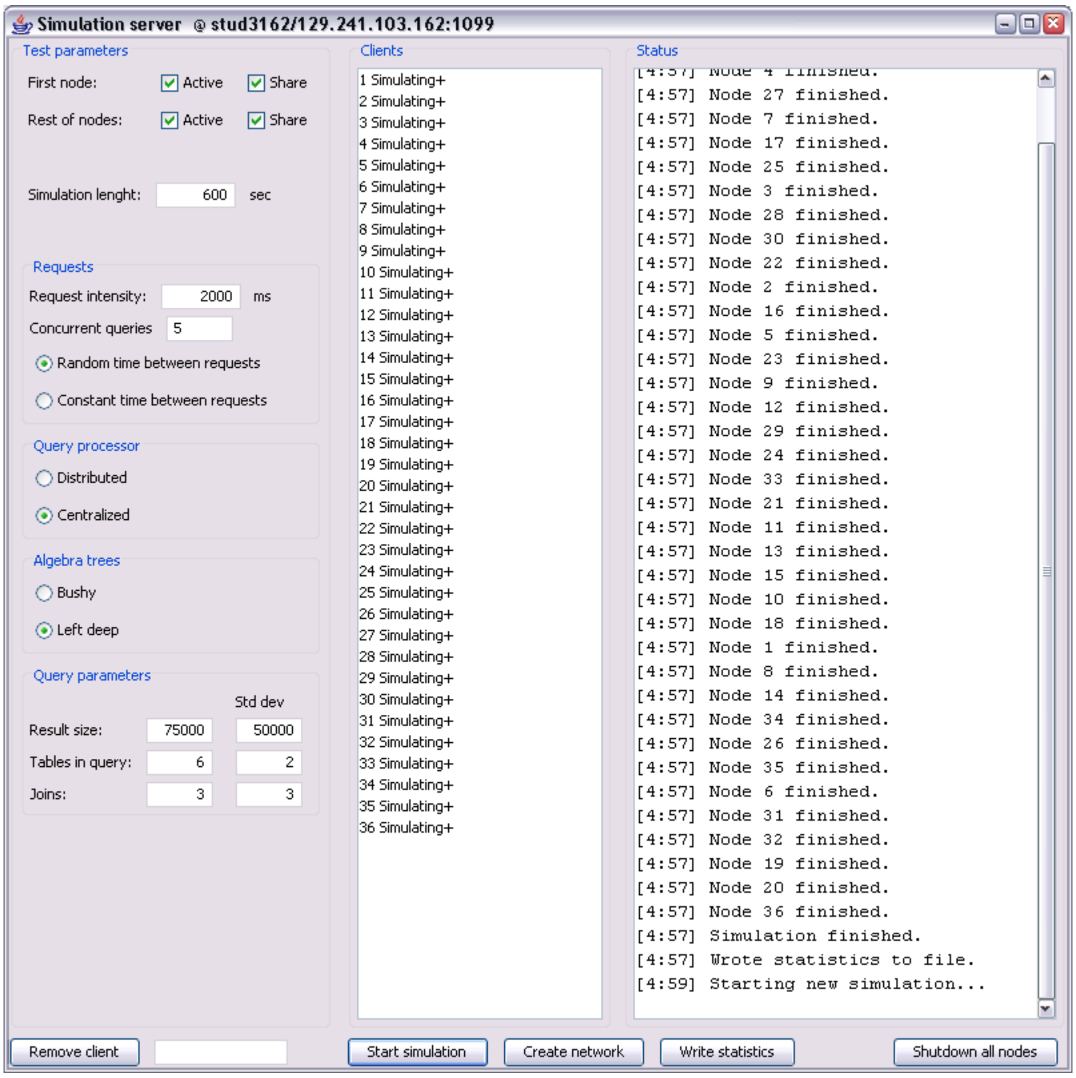


Figure 5.1: Test architecture

The test application initially connects only to the test server. The server then sends parameters such as the number of tables each node should store and the number of tuples in each table. The test server can order nodes to join the DHT at any time, the first node that joins will create a new ring. When a new simulation starts, each node will receive simulation parameters and an overview of which tables it can include in its queries. The test application will generate queries at a specified rate while the simulation is running. There is no communication between the test server and the nodes during this step. Nodes send their statistics to the test server after the simulation is done. It is possible to start new tests with differnet parameters without restarting the system. One should note that some parameters, such as database size, cannot be changed from one simulation to another without restarting the entire system.

The communication between the nodes and the test server is done by a single TCP connection per node. This will allow nodes to communicate with the test server even if they are behind a firewall that blocks incoming connections (the connection is an outgoing connection). Last years test environment relied on Java RMI and required a connection in each direction to be initialized. This made it impossible to test PORDaS on UNIX clusters.



First node: [✓ Active] [✓ Share]

Rest of nodes: [✓ Active] [✓ Share]

Simulation lenght: 600 sec

Requests

Request intensity: 2000 ms

Concurrent queries 5

[✓ Random time between requests]

Constant time between requests

Query processor

Distributed

[✓ Centralized]

Algebra trees

Bushy

[✓ Left deep]

Query parameters

		Std dev
Result size:	75000	50000
Tables in query:	6	2
Joins:	3	3

```
1 Simulating+
2 Simulating+
3 Simulating+
4 Simulating+
5 Simulating+
6 Simulating+
7 Simulating+
8 Simulating+
9 Simulating+
10 Simulating+
11 Simulating+
12 Simulating+
13 Simulating+
14 Simulating+
15 Simulating+
16 Simulating+
17 Simulating+
18 Simulating+
19 Simulating+
20 Simulating+
21 Simulating+
22 Simulating+
23 Simulating+
24 Simulating+
25 Simulating+
26 Simulating+
27 Simulating+
28 Simulating+
29 Simulating+
30 Simulating+
31 Simulating+
32 Simulating+
33 Simulating+
34 Simulating+
35 Simulating+
36 Simulating+
```

```
[4:57] Node 4 finished.
[4:57] Node 27 finished.
[4:57] Node 7 finished.
[4:57] Node 17 finished.
[4:57] Node 25 finished.
[4:57] Node 3 finished.
[4:57] Node 28 finished.
[4:57] Node 30 finished.
[4:57] Node 22 finished.
[4:57] Node 2 finished.
[4:57] Node 16 finished.
[4:57] Node 5 finished.
[4:57] Node 23 finished.
[4:57] Node 9 finished.
[4:57] Node 12 finished.
[4:57] Node 29 finished.
[4:57] Node 24 finished.
[4:57] Node 33 finished.
[4:57] Node 21 finished.
[4:57] Node 11 finished.
[4:57] Node 13 finished.
[4:57] Node 15 finished.
[4:57] Node 10 finished.
[4:57] Node 18 finished.
[4:57] Node 1 finished.
[4:57] Node 8 finished.
[4:57] Node 14 finished.
[4:57] Node 34 finished.
[4:57] Node 26 finished.
[4:57] Node 35 finished.
[4:57] Node 6 finished.
[4:57] Node 31 finished.
[4:57] Node 32 finished.
[4:57] Node 19 finished.
[4:57] Node 20 finished.
[4:57] Node 36 finished.
[4:57] Simulation finished.
[4:57] Wrote statistics to file.
[4:59] Starting new simulation...
```

Figure 5.2: The test server application.

5.3 Tests

This section presents the contents and purpose of the two tests that were planned.

Test Parameters

Many of the parameters in the two tests were common. The reasoning behind the setting of parameters was to simulate a medium load and a probable usage pattern. Databases are kept small since the desire is to test PORDaS and not Derby. The time between requests is Poisson distributed with a given mean value. The queries will wait if there are more concurrent queries than the maximum number of concurrent queries allowed.

- Tables per node: 8
- Tuples per node: 2000
- Number of nodes: 36
- Simulation length: 10 minutes
- Request intensity: 2 seconds
- Random time between requests
- Joins in query: 3 (deviation 2)
- Number of tables in query: 4 (deviation 2)
- Number of concurrent queries: 5
- Maximum result size: 100 000 (deviation 90 000)

Test 1

In the first test, all nodes were sharing and active. The purpose was to compare every permutation of type of algebra tree and type of planner, which means that 4 simulations were be run. The interesting data in this case are the number of started queries versus the number of finished queries, and the response times for each permutation.

Test 2

The purpose of the second test was to study and isolate the effect of executing queries in parallel. This was achieved by having one active, non-sharing node query the rest of the nodes, which were sharing and inactive. The only task of the non-active sites in the system was resolving the queries the one active node gave them. Two tests were run. The first test had the query processor make linear trees and used a centralized execution strategy. The second test used bushy trees and a distributed execution strategy.

5.4 Results

This section presents and comments the results of the tests.

Test 1

Figure 5.3 shows a comparison of all permutations of type of query planner and type of algebra tree with respect to the number of started and finished queries. In both cases using centralized plans the loss of queries is minimal, while in the distributed case, the loss is noticeable. With a linear query tree, the loss is over 13%. The circumstances indicate that the reason for the loss is that queries timed out before results were received. The time-out threshold was set to two minutes. By looking at the maximum response times for the queries that did finish, these are close to this limit.

The same figure also shows a distinct difference between the centralized and distributed simulations. The number of started queries is much lower in the distributed case. The reason for not starting more queries is because of the limit on the number of concurrent queries. Waiting for a query to be executed has the effect of lowering the throughput.

Figure 5.4 and figure 5.4 show the response times for all permutations of query planner and algebra tree. Times are given for when the first, tenth, hundredth and so on tuple was received. The series in the graphs are divided into categories based on the size of the result.

Similar for all the simulations is that for the category of results of size 0-100 tuples, there is a decrease in response time from when the first to the tenth tuple is received. This means there are queries with less than 10 tuples in the result that take a long time to execute. The reason why it takes more time to execute might be that the queries are more complex, containing more joins and selections. The size of the result does not say much about the complexity of the query.

In the distributed, bushy case and in the centralized, bushy case, the category of queries with results in the range of 250-500 tuples have high response time compared to the others. This series is based on scarce data, with only 11 queries. The maximum response time for this series is high, with over 104 seconds. The next range has maximum response times of 11 seconds, so the anomalous range is likely to consist of outliers.

In both the distributed cases, the largest range of results has significantly lower response time than many of the lower ranges. The same goes for the two second highest and third highest ranges as well. It is probable to assume the explaination lies with the way queries are formed by the simulator. Basically, the maximum result size is determined in advance for each query created. To get smaller queries, joins are added to constrain the size. When large queries occur, they have many cartesian products[6]. Cartesian products are always exe-

[6]The number of tables in a query is determined per query, and if there are few tables available, the only way to get big results is through cartesian products.

cuted locally, which means that smaller parts of the query is delegated to others, which again translates to less network traffic.

Both the centralized simulations have a noticeable artifact. The series with results in the range of 1000-2500 tuples have a terrible response time for the 1000th tuple. It is hard to say why this is so, as the statistics gathered is too coarsly grained to identify the source of error.

When comparing the centralized and the distributed simulations, it is obvious that the centralized plans perform much better. The second test was designed to study this closer.

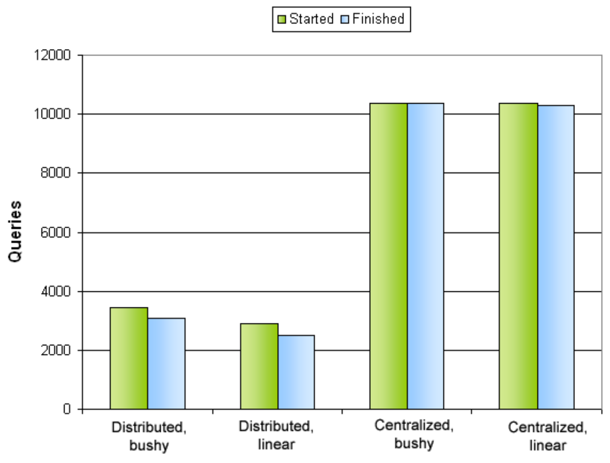


Figure 5.3: Started versus finished queries

Test 2

Figure 5.6 shows a comparison of the minimum, average and maximum response times observed for queries with a resultsize between 25.000 and 50.000 tuples. The distributed execution strategy performs consistantly worse than the centralized strategy.

Discussion

PORDaS was able to conduct the simulations without node failures. This does not prove the correctness or scalability of PORDaS, but it shows that the system is stable enough to handle moderate work loads between a modest number of peers.

Both tests show that the centralized execution strategy is better than the distributed. The distributed execution strategy has the advantage of executing operators in parallel, but failed since the joins always had a huge selection rate. This caused the distributed execution strategy to generate a lot more network traffic than the centralized strategy. The distributed strategy could have worked

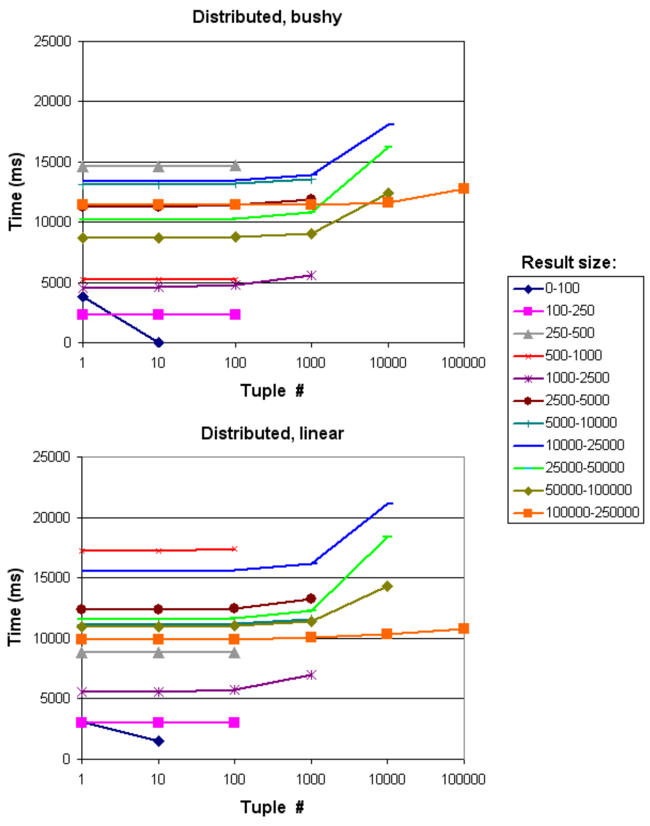


Figure 5.4: Average response times for distributed query plans

better in comparison to the centralized strategy if the joins had lower selection rates.

The tests show the importance of choosing the correct heuristics when processing queries without an optimizer. A better heuristic would probably be to only perform equi-joins in a distributed fashion by using an operator like the pipeling hash join or hash ripple join. The other joins could be centrally performed with the current join algorithm.

The test application also deserves some commenting. Classifying queries by the size of the result does not reveal much about the complexity of the queries. Generating queries at random did also make it difficult to identify specific problems. It would probably have been better to use a small set of pre-defined queries to get better control over the work load. More extensive logging of events could help identify problem areas, but probably at the expense of the performance.

An additional problem with the statistics is that some of the tests contain too few samples for some query classes. This is due to the short simulation time and the fact that the test application did not have much control over the final result size. This could be solved easily by running longer tests, but due to the short period of time available at the computer lab the simulation times were kept short.

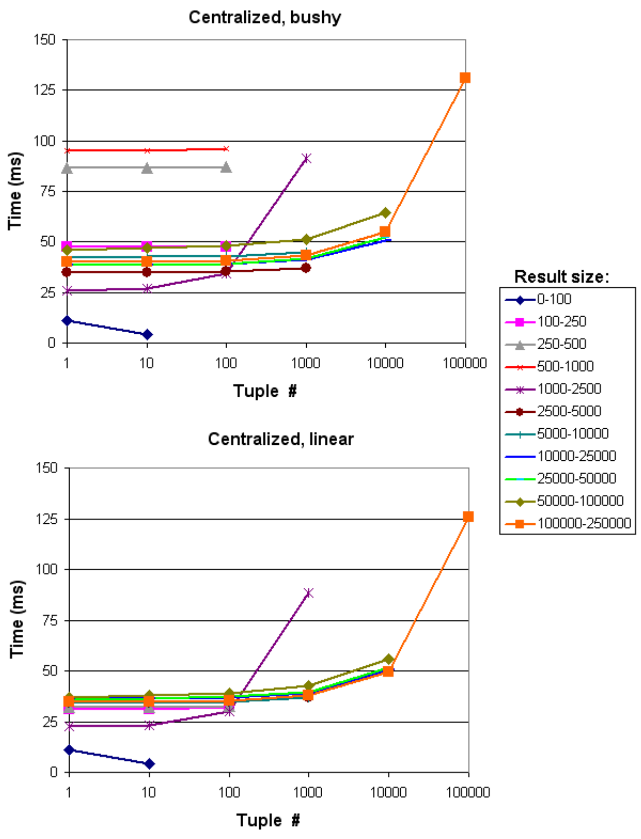


Figure 5.5: Average response times for centralized query plans

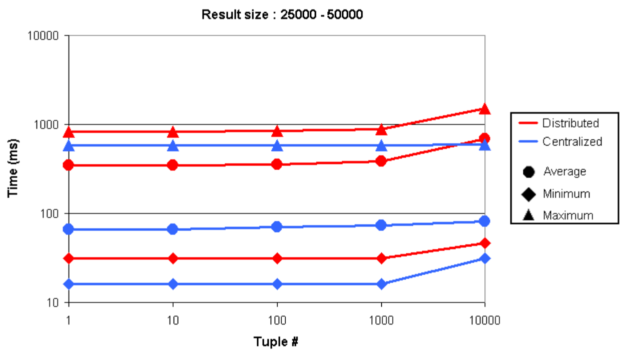


Figure 5.6: Comparison of distributed and centralized query processing

Chapter 6

Conclusions and Further Work

A major part of the master thesis has been searching for and studying material related to P2P systems and databases in a distributed environment. There is a lot of theory to cover, and a great deal of effort has been put into acquiring an overview of challenges and opportunities.

The next part was using the gained knowledge to propose a system which integrated the fields of P2P technology and databases into a P2P database. The central area of focus was the query processor, and finding paths for optimization. The most promising ideas were using a dynamic query processor, forcing relaxed consistency and introducing a scheme for replication.

The foundation from the project of fall 2005 was extended by incorporating a pipelined, distributed query processor. Finally, there was a round of simulations that proved the system could be deployed and run on a modest cluster of 36 computers. The simulations showed that the centralized query planner performed much better than the distributed one. It indicates that the heuristics employed were not optimal.

However, several of the features displayed in the analysis were left out of the implementation due to time considerations. Developing the required for distributed query processing foundation took longer time than expected, and it became apparent that creating a query processor, despite numerous simplifications, is a complex task. Adding a P2P aspect to the system did not make it any easier, on the contrary. Distributed computing is also complex, which was quickly discovered when trying to manage a heavily multi-threaded program. It lead to the conclusion that a single-threaded design would be more appropriate.

The scope of the master thesis was broad from the beginning, opening for more than one direction. As there was no available P2P database framework to use as a starting point, this had to be built before any specialized inquires could be mounted. It soon became apparent that each component in the framework was comprehensive enough to constitute a master thesis in themselves.

The bottom line is that P2P databases with relaxed consistency is a promising architecture, and that multiple opportunities exists for optimization.

The future work on PORDaS could take several directions but the most natural progression would be to improve the query processor. This includes adding an appropriate join algorithm for equi-joins, implementing query operators like iterators instead of threads and adding a form of distributed eddies to do query optimization.

With a proper query processor in place a replication scheme (section 3.3.1) could be introduced to increase performance and achieve better load balancing. Another important improvement would be to increase the robustness of PORDaS. There has not been much focus on how to deal with departing nodes and errors during query execution. The current query processor relies on timeouts, but this could be wasteful if there are available replicas.

PORDaS does not currently possess many of the features found in modern a ORDBMS. The query processor should be expanded to be able to perform aggregations on data, set operators, nested queries etc. There is also a lot of work left to be done on the object relational features as the current implementation of abstract data types merely serves as a proof of concept.

Appendix A

Pastry

This appendix presents Pastry [16], which is a DHT that adapts to network locality.

Topology

Pastry arranges its nodes in a 128-bit one-dimensional circular identifier-space. Each node in the Pastry network is assigned a 128-bit node identifier (nodeId) by computing a cryptographic hash of the node's IP address or public key. For the purpose of routing, nodeIds and object keys are thought of as a sequence of digits with base 2^b. Routing is performed as follows. In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's id.

In order to support the Pastry routing procedure each node maintains a routing table. The routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries in each. The 2^b-1 entries at row n of the routing each refer to a node whose nodeId shares the present node's nodeId in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n+1$th digit in the present node's id. Each entry in the routing table is only one of many possible nodes. In order to provide good locality properties the nodes are chosen according to a proximity metric. See figure A.1 for an illustration[1].

Each node does also maintain a leaf set and a neighbor set. The leaf set consists of the most immediate neighbors in the nodeId space. The neighbor set contains the nearest nodes in terms of the number of IP routing hops. The purpose of the neighbor set is to ensure that Pastry can take network locality into account when choosing routing table entries. See figure A.2 for an illustration [2].

[1]Adopted from [16].
[2]Adopted from [16].

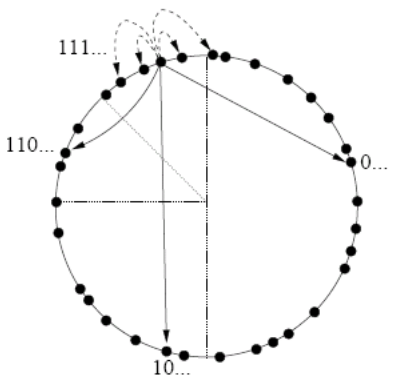
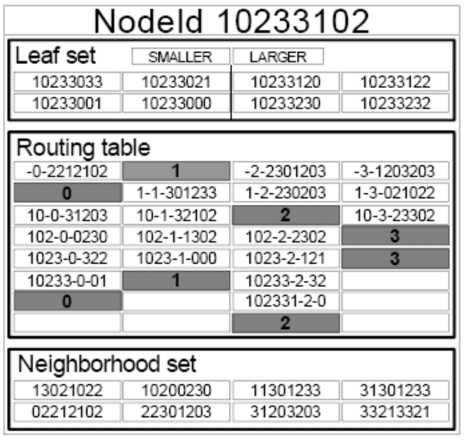


Figure A.1: The Pastry overlay network


NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure A.2: Routing table in Pastry

Lookup

The principle behind Pastry's routing procedure is to always send a message to a node whose nodeId is numerically closer to the given key than the current node's nodeId.

If the key falls within the leaf set, the message is forwarded directly to the destination node. If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit. If no such node is reachable the message will be forwarded to a node that is numerically closer to the key.

Using a deterministic routing procedure makes the system vulnerable to failed nodes. If a node accepts messages without forwarding them correctly, repeated queries will fail every time. In applications where such node failures must be tolerated, the routing can be randomized.

Maintenance protocol

If a node in the routing table or leaf set fails, it has to be replaced by another node. The replacement is found by contacting nearby nodes.

Each node in the neighborhood set is contacted periodically to see if it is still alive. If node is not responding, a replacement will be found through other neighbors' neighborhood sets.

Replication

Replication in Pastry is done by replicating objects on the k nodes with the numerically closest nodeIds to a key in the nodeId space.

Appendix B

The PORDaS Query Language

This appendix defines the query language used in PORDaS using Backus-Naur form.

sql_command ::= select_command | insert_command | update_command | delete_command

select_command ::= ”select” [”all” | ”distinct”] ”*” | column {”,” column}
”from” table { ”,” table } [”where” logical_term]

column ::= {table ”.”}identifier

table :: = identifier

identifier ::= ”letter” { ”letter” | ”digit” }

logical_term ::= logical_factor { ”and” logical_factor }

logical_factor ::= expression comparison_op expression

comparison_op ::= = | < | >

expression ::= constant | column

Appendix C

Messages

This appendix presents the messages used in the communication between nodes in a PORDaS network. Figure C.1 illustrates all message types. These are categorized as keywords, tables, queries and table definitions.

Keywords

Creating, maintaining and querying the keyword index requires 5 types of messages.

The *keyword insertion message* is used to insert keywords to the index. This is done when the node connects, or upon request. The hashed keyword is the key for the record, used by the index to identify the value. The table definition is the table associated with the keyword.

The *keyword refresh message* is used to refresh a keyword record. Both the hashed keyword and the hashed table definition must follow the message, to specifically target the correct record. This is because two different table definitions may be described using the same keyword. When a refresh message ends up at a site that does not have that keyword in its index, the keyword record must be requested. The from field is used for this purpose. If the node responsible for a given keyword record goes down, the refresh message will end up at the next node in the DHT.

The *keyword index record request message* is used to request a specific keyword record. When this message is received, it forces the node to send a *keyword insertion message.*

The *keyword request message* is used to request all table definitions related to the given keyword. The from field is used when returning the response.

The *keyword response message* is sent to answer a keyword request. It contains all table definitions that are described with the specified keyword.

Tables

PORDaS relies on the index to store the location of tables. Three messages are used to serve this purpose.

The *owner insertion message* is used to add an owner of a specific table to the table index. All nodes periodically send one such message for each of the tables in their local database. There is no table refresh message because it would contain the same fields as this message.

The *owners request message* is used to request a list of all the locations where the specified table can be found. The from field says where to return the response.

The *owners response message* is sent when responding to table requests. It contains all the locations of the requested table.

Queries

In order to fetch tuples and delegate responsibility for a sub query, two messages are needed.

The *sub query request message* is sent to ask a node to resolve the given query tree. The external id is needed in order to know at which point in the query tree the response should be inserted.

The *sub query response* is a tuple with an external id. This id plugs the tuple in at the right place at the requesting node.

Table Definitions

To answer queries that reference tables using their hashed table definition value, the index must be able to find out which table this is, and where it is stored. This requires two messages.

The *table definition request message* is sent to retrieve the table definition connected to the hashed table definition. The from field is needed in order to know where to send the response.

The *table definition response* is sent as an answer to a table definition request. It contains the requested table definition.

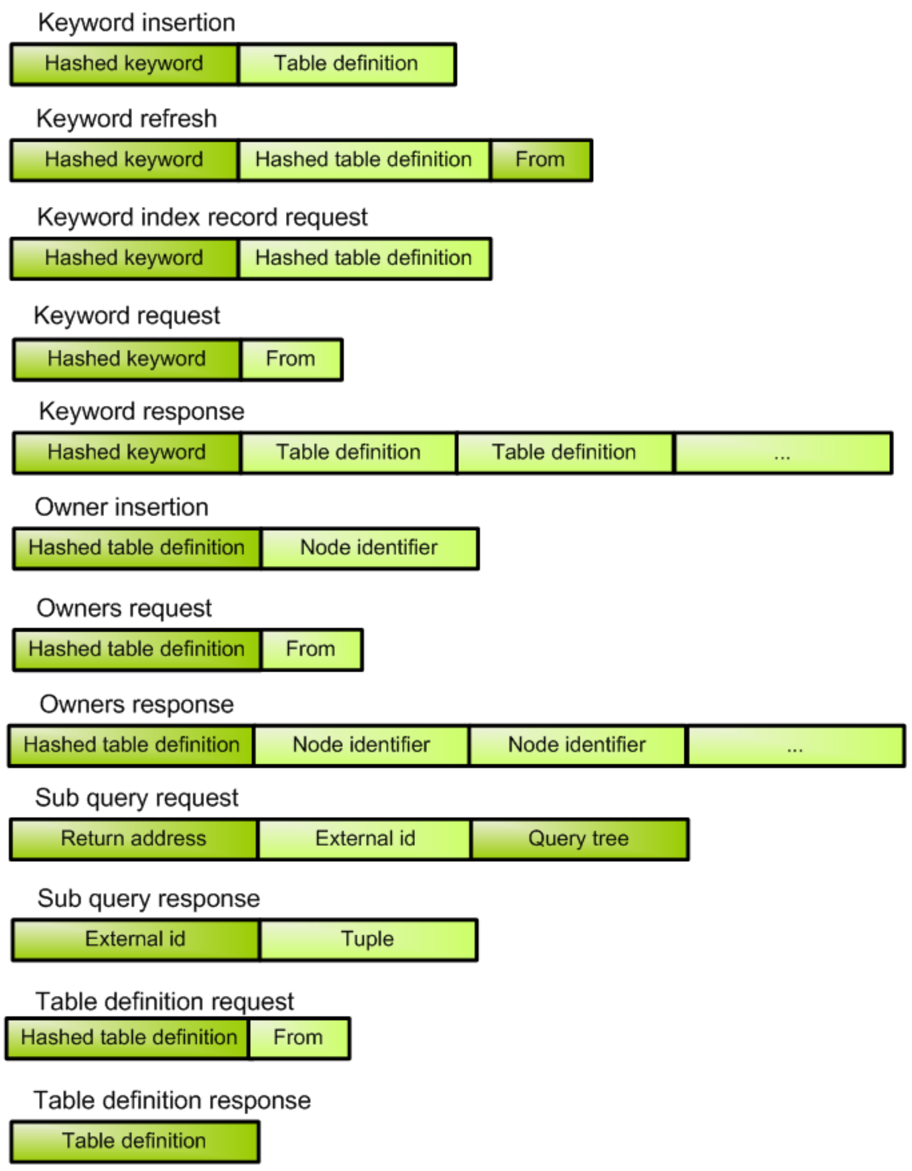


Figure C.1: PORDaS messages

Bibliography

[1] Thomas Connolly and Carolyn Begg. *Database Systems.* Addison-Wesley, 2002.

[2] M. Tamer zsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second edition.* Prentice Hall, 1999.

[3] Stonebraker, Michael Hellerstein, and Joseph M. Hellerstein. *Distributed Database Systems, Readings in database systems.* Morgan Kaufmann, 1998.

[4] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. 1998.

[5] Stonebraker, Michael Hellerstein, and Joseph M. Hellerstein. *Objects in Databases, Readings in database systems.* Morgan Kaufmann, 1998.

[6] Karl Aberer and Manfred Hauswirth. Peer-to-peer information systems: concepts and models, state-of-the-art, and future systems. 2002.

[7] Jordan Ritter. Why gnutella can't scale. no, really. 2001. http://www.darkridge.com/ jpr5/doc/gnutella.html.

[8] K. Gummadi, R. Gummadiy, S. Gribblez, S. Ratnasamyx, S. Shenker, and I. Stoicak. The impact of dht routing geometry on resilience and proximity. 2003.

[9] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. 2004.

[10] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. 2004.

[11] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM'01*, 2001.

[12] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, 2002.

[13] H.V Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. 2005.

[14] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier.

[15] The freepastry dht, rice university. http://freepastry.rice.edu/FreePastry.

[16] A. Rowstron and P. Druschel. Pastry: Scalable distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[17] Mysql database. http://www.mysql.com.

[18] The apache derby database project. http://db.apache.org/derby/.

[19] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[20] R. Huebsch, B. Chun, J. M. Hellerstein, B. T. Loo, , P. Maniatis, T. Roscoe S. Shenker, I. Stoica, and A. Yumerefendi. The architecture of pier: an internet-scale query processor.

[21] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. Objectglobe: ubiquitous query processing on the internet.

[22] Michael Stonebraker, Paul Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. 1995.

[23] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS*, pages 242–259, 2002.

[24] A. Philip, G. Fausto, K. Anastasios, M. John, S. Luciano, and Z. Ilya. Data management for peer-to-peer computing: A vision. in webdb workshop on databases and the web, 2002.

[25] P. Boncz and C. Treijtel. Ambientdb: relational query processing in a P2P network.

[26] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing.

[27] Kjell Bratsbergsengen. *Lagring og behandling av store datamengder.* Tapir akademisk forlag, 1997.

[28] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, 1979.

[29] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallell main-memory environment. 1991.

[30] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[31] Ryan Huebsch and Shawn R. Jeffrey. Freddies: Dht-based adaptive query processing via federated eddies. Technical Report UCB/CSD-04-1339, EECS Department, University of California, Berkeley, 2004.

[32] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[33] Y. Zhou. Adaptive distributed query processing. In *VLDB*, 2003.

[34] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[35] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298, 1999.

[36] Venugopalan Ramasubramanian and Emin Gn Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays.

[37] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[38] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001.

[39] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. *icdcs*, 2004.

[40] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 1992.

[41] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. 2003.

[42] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. 2004.

[43] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM'01*, 2001.

[44] A. Guttman. R-trees: A dynamic index structure for spatial searching. 1984.

[45] Erik Buchmann, Klemens Bohm, and Otto von Guericke. How to run experiments with large peer-to-peer data structures. 2004.

[46] Sean Rhea. The bamboo dht. http://bamboo-dht.org/.

[47] The planetlab. an open platform for developing deploying and accessing planetary-scale services. http://www.planet-lab.org/.