Armada
An Evolving Database System

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op woensdag 10 juni 2009, te 10.00 uur

door

Fabian Emiel Groffen

geboren te Haarlem

Promotiecommissie:

Promotor:	prof. dr. M.L. Kersten
Copromotor:	dr. S. Manegold
Overige leden:	prof. dr. P. Valduriez
	prof. dr. M. van Steen
	prof. dr. P. Klint
	prof. dr. B.J. Wielinga

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research reported in this thesis was carried out at CWI, the Dutch national research centre for mathematics and computer science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

SIKS Dissertation Series No. 2009-18
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research and implementation effort reported in this thesis was partially funded by BSIK/BRICKS within the BRICKS IS2 project.

ISBN 978-90-9024262-0

Contents

1 Introduction

1.1 Background

The era of huge monolithic main-frames is over. Instead, server rooms are now populated with clusters of machines that serve the workload. No longer a single system, but instead a group of systems is responsible for a particular task. Clusters typically have a bigger combined capacity that can easily be expanded by adding new systems. Having multiple systems being responsible, failure of an individual system does not stop the system as a whole, but merely leads to a graceful degradation.

Not surprisingly, there is a trend to deploy for almost every task where a computer is involved, a cluster setup involving multiple machines. However, not every task can as easily be deployed on a cluster as others. Typically, informative web-sites such as those of search engines, can be served off any server that has a replica of the original server's contents. Periodical updates to the contents are easy, as none of the replicas change the data, only a designated "master" performs updates. These replication clusters are well suited for *read-only* tasks, however, as soon as an application generates a heavy, interactive *write* workload, simple replication techniques are no longer sufficient. Consider a database used by a web store to record what each customer ordered. While e.g. the catalogues and pictures on the web sites themselves may be easily replicated, actual customer orders cannot since those are typically frequently updated. To use a cluster for the shopping data, either the replication techniques need to ensure that all replicas in the cluster have the same order data at the same time, or fragmentation has to be used to split the data up over the cluster, where each system in the cluster is responsible for its own part of the data. These two techniques, in particular for storing and retrieving of data, are deployed in database technology to benefit from a cluster setup.

Distributed *Database Management Systems* (DBMSs) cover the area of deploying databases on a cluster. They *store* structured data. Replication and fragmentation are well researched topics and various forms of either copying or splitting the database contents exist. However, the nature of a DBMS is to be well *in control*. Certain properties that equip a distributed DBMS with robustness, reliability and correctness (referred to as ACID properties) prevent the individual systems from operating on their own. Instead, they have to *serve* the system that coordinates the process of retaining the ACID properties in the cluster. This PhD thesis explores the setting in which a distributed DBMS seeks to loosen the ACID properties to the degree where in contrast to traditional clusters, the distributed DBMS consists of sovereign and autonomous database systems. The autonomy of the systems allows them to solve problems on their own, using local information to influence the solution.

In a search for structured access to the data, database technology has been developed over the years. Not only the data volumes grow, but also the use of databases for storing that data. Stimulated by freely available database servers of all kinds, applications are (re-)designed to use databases as their storage back-ends. For some applications not only the data, but also the state of the application is stored in a database to benefit from their reliability and persistence, like in Hibernate [39]. Databases are more and more often used, with an increasing amount of data involved.

In Werner Vogel's keynote at VLDB'07 [72], database scalability is identified as an indispensable ingredient required by applications of world wide sizes. The matter in which a database can be extended to scale to a new level, defines its usefulness to the application in terms of adapting to its data needs. This process, where a database *grows* along with the data requirements of the application, is coined as *incremental scalability*. The database *evolves* over time, requiring a flexible administration of data whereabouts to support reconfigurations. In the light of this property, autonomy is an *enabler* for flexible reconfigurations. No longer a change in the system needs to be agreed upon by the distributed DBMS, but just the local system that is involved takes care of the operation, whenever appropriate.

Such freedom resulting from autonomy needs a strategy to become effective in a cluster. The generic model proposed in this thesis is aimed at administering data spread over multiple autonomous systems. The administration includes generic functions that define how the data is spread. These functions are independent, and hence their implementation is decided upon by the system involved. With specific implementations of those functions, existing distributed database systems can be emulated, thereby it demonstrates the model's generic nature. Being a model to define how original, central, data is spread around, we acknowledge that it need not to be specific to relational databases only. However, we approach the model from a relational database point of view.

In this thesis we study the distribution of databases with evolutionary behaviour as focus. To achieve the flexibility of this evolution process, we research a database system comprising autonomous systems. With this study we make steps into directions for further research and development of decentralised, evolving and autonomous database systems.

The ability to adapt to the change in data requirements is considered important and an enabling technology that is necessary to deal with the future. The abilities to flexibly and sufficiently adapt to data requirements are missing in many database products today.

1.2 Research Questions

The focus of this thesis is an exploration towards *autonomy*, *decentralisation* and *evolution* of database systems. The general research question addressed, stresses the continuous aspect of evolution in a dynamic database:

> *How to support a continuously evolving database management system consisting of autonomous sites and a decentralised catalog?*

The research question defines decentralisation and autonomy as important directions of the exploration. To get a better grip on how to answer this question, we refine the general research question into four more specific questions. The first question addresses data distribution with the autonomy and derived independence of sites that are part of a larger database management system:

> *1. In what way can we distribute data in a dynamically evolving system using site local decisions and avoid global site control?*

The aforementioned site autonomy has effects on how those sites can be used by clients. In particular the question on the level of client participation is formulated by the next question:

> *2. What is the role of application clients in an autonomous, distributed database management system?*

Tightly related to the autonomy dimension is the evolution component. The next question expresses the focus on the continuity of that evolution and how to make it an integral part:

> *3. How can incremental scalability become a natural component in an evolving system?*

Finally, in a search for how our ideas affect implementations of distributed database systems, we investigate the effects of our exploration on existing machinery in the last question:

> *4. To what extent are existing common techniques to manage a catalog sufficient to support autonomy, decentralisation and evolution?*

1.3 Approach and Outline

To answer the research questions, we start with the introduction of a model in Chapter 3 that forms the foundation for all the subsequent chapters of this thesis. The formal model describes a way to administer the evolution process of a database. The model itself has emerged in an evolutionary process of many iterations, with each iteration being a refinement based on new insights gained during the process. Eventually, a generic, minimal form of the model was found, which is described in Chapter 3. The assumption of the use of autonomous local decisions made by the participating systems is embedded in the model. With this assumption, the model — as heart of this very thesis — answers the first research question by describing how data distribution can be done using autonomous participants. The model also answers to a large extent the third research question, because its design includes continuous refinement, empowered by the autonomy of sites to locally reconfigure and refine.

The model achieves incremental scalability by operations applied to the data. These operations can contain an arbitrary function working on the data. In Chapter 4 we examine these functions, their properties and their effect when used in the model. The operations applied within the model allow for continuity during the evolution of the database, and hence form the rest of the answer to the third research question.

The autonomous nature of the model directly affects its application clients. Chapter 5 describes an autonomous query strategy that matches the model, thereby answering the second research question. The autonomy used by the strategy makes it unconventional and yet largely unexplored territory. In Chapter 6 we explore the effects of the model combined with different variants of the application client strategy to gain some insight on the viability of both when put in practice.

To answer the fourth question, Chapter 7 presents a simulation study, in the de-facto query language SQL. This study includes a little experimentation on the overhead of this implementation on some SQL database systems. It is shown that standard DBMSs are far from being able to support the proposed model. Therefore in Chapter 8 we describe the last contribution of this thesis, an architecture that supports the model and strategy that are part of our exploration.

Before we start with previously mentioned chapters, in the next chapter we first give an overview of work done in the field. It positions this thesis in the broad area of database research. The overview is followed by the main chapters of this thesis, which in turn are followed by a conclusion and outlook on future work. An appendix on developed technology aimed at a realisation of the ideas presented in this thesis is included for background information purposes.

2 Overview

2.1 History

It is the nature of humans to *organise*. Admittedly some more than others, but deep inside we have a force that makes us form groups, build houses in orderly streets, define time tables for public transportation, *etcetera*. A place for everything and everything in its place. Most of our organisational efforts lead to more efficiency afterwards. Many streets are clustered around a given theme, in a city district, of a certain city, in a province, state, country and so on. But most of all, house numbers in a street are *ordered* ascending walking away from the city centre, one side the odd numbers, on the other the even ones.

Perhaps the most obvious example of our organisation drive is the library. On many shelves, per genre, all books are ordered first on author, then on title, just to achieve an easy search process afterwards. But, libraries are old fashioned and books are made of paper — a material that can only be used once and worst of all, tends to decay over time. No matter how well the library is organised, going to the library takes time. Walking around to find a book takes a few precious seconds, and do not forget the browsing through the book itself. Over the years, driven by another nature, man has found his way into the digital era. Even though this era is mainly *virtual*, the organisational drive has found its way deep inside it. As a world solely created by man, organisation is the key component it is made out of. Everything is nicely addressable, no exceptions for which no proper rule is defined, no chaos that eventually does not appear to be a well organised structure with logical rules. Among other things, organised storage of data in this organised digital world is nothing more than a logical thing to do, and hence the *digital* databases were born.

In the grand scheme of things, databases have a long history, even going back centuries if one considers the non-digital libraries and records as a *data base*. Nowadays' digital libraries certainly show much resemblance in the storage and search process with those non-digital libraries. Starting from the 1960's, computers became cost effective, and database technologies were developed. The new databases were used to store administrative data, such as flight reservations. These were not books, but these *records* were searched for, like books. For this a similar structure as found in libraries was necessary, including a method to conveniently search for those records. Codd laid the foundations of modern databases with a proposal for a relational model [16] around 1970. In this model, the logical organisation of the database is disconnected from the physical organisation. Data comes in relations that can be considered to be sets, inheriting set operations. This yielded in relational queries over the data. The proposal by Codd resulted in two proofs of concept: UCB's Ingres [66] and System R [4] from IBM. The latter used a query language SEQUEL, whose descendant SQL became the *de-facto* database query language in the mid 1980's.

2.1.1 Distributed Databases

Soon these databases were extended to run on multiple machines instead of on a single monolith [33]. The relational model suits fine in this distributed setting which goes one step further in decoupling the logical organisation of a database from the physical storage, by hiding the location of the data from the user. As such, the distributed variant is a *drop-in* replacement for the single database system. Three architectures for distributed databases exist [68] with either a component *shared* or nothing in common with the other systems. *Shared disk* and *shared memory* systems are architectures where multiple CPUs work with a single disk or memory subsystem. In the shared memory architecture, all processors and disks share a common, often relatively large, memory. Communication between processors can hence go through reads and writes in the shared memory. In the cluster approach of shared disks, each processor has its own memory, but access to a single large disk. The *shared nothing* variant has no hardware components shared. Instead, separate systems send each other messages through a network between them. Finally a hybrid combination of these architectures can be formed into a hierarchy.

Network The shared nothing case has become popular, due to its cost effectiveness [63]. However, in the early days, this architecture was hindered by network instability, seriously affecting the process of shipping several megabytes of data per day. Hence, only once the network connections became fast and reliable, this architecture became feasible and needed. Distributed shared nothing databases are scalable systems, consisting of multiple standard computer systems which are more powerful and cheaper than a big mainframe computer [18]. This trend has been stimulated by the rise of the Internet, which is mainly built of network-connected shared nothing systems.

Parallelism The distribution, which was originally intended for a shared data base between geographically spread offices, allows for parallelism [18, 33] as well. When exploited well, this parallelism can result in considerable performance improvements. Hence, the query processing on this architecture tries to exploit this. In the Volcano model [27] the query processing design is modelled in such a way that it can be easily switched to parallel execution. The “brackets” of this model are arbitrary operations based on one or more input channels with as result one output channel. These brackets can be placed on other machines without changing the execution plan. Since shipping large amounts of data over the network is inefficient, this should be taken into account when doing distributed query processing. It is represented by the query shipping versus data shipping approaches to the data processing problem. Factors which influence the decision for either shipping the full data to the processor or first doing the processing where the data is and shipping that result include network characteristics such as speed and latency, the processing power of the involved systems and whether the system doing the processing has all the other data available to do the processing [46].

Heterogeneous and Federated Databases For various reasons, companies and institutions end up with several types of databases, often from different vendors. These heterogeneous databases are usually incompatible with each other and need additional measures to still perform joined work [24, 43]. This is the area of federated and heterogeneous database systems. In those systems, a *mediator* delegates full or partial queries to the underlying database systems via a wrapper that encapsulates the database specific behaviour into a generic request and response [32]. Because the capabilities of each database are generalised this way, the possibilities are limited [43].

Data Placement Where the data resides partially defines where it can best be queried. Static data placement, assumes the data to be positioned as defined by a database administrator [53]. Data location then depends on what kind of queries take place from what locations in the system [12]. Tools to examine the query workload function as input on where to place what data. However, the real query workload, is hard to predict, as well as tends to change over time. Relocation of the data by the administrator is a tedious job, which triggered a focus towards dynamic data placement approaches [35]. The ideal here is to keep statistics and automatically move and make copies of data to adapt to the current workload [53, 11]. Work has been done on replication and caching of data to dynamically match the workload. Here two types can be distinguished. First the algorithms that try to reduce the communication costs by placing the data close to the clients that are likely to use the data e.g. by static prediction of a workload, and second the algorithms that do load balancing by replicating popular *hot* data dynamically [61, 9]. Data moving is an extension to replication, where a final migration stage abandons the old copy, such that the new copy on its new location becomes the primary one.

Replication With multiple copies of the original data, updates become less trivial to apply. Depending on the level in which all replicas need to be consistent with each other, architectures were designed [68]. An approach to keep consistency throughout all replicas is to force all copies to apply the same update at the same time. This is an expensive method as it requires all replicas to make the update disallowing any other operations on that data at the same time. Techniques to only use a majority of the replicas, a quorum [25, 69, 23], were devised to relax this requirement. A hierarchical architecture typically has a primary copy, on which all updates are performed. All other replicas receive the updates from the primary copy. The other copies may be out of date, but still consistent for what they store. Since updates are only done on the primary copy, no conflicts can arise. However the system may fail completely when the primary copy fails, hence techniques to choose a new master copy dynamically aim to resolve those cases. Other approaches use asynchronous updates to replay update statements when necessary to defer doing the work until really necessary.

Economic Clients In a system where clients can choose between multiple servers to execute their query on, the complexity of computing the right decision becomes too large for a single system to control. Systems relying on “economic models” eliminate this complexity via bidding for resources in an auction. Each server offers certain services for a given “price” as a response to a client’s query. The first proposed distributed database system based on this economic drive, was Mariposa [65]. In this system, clients assign a budget for each query. Brokers in the system start auctions to provide the necessary data and to perform the required operations on this data. Servers that are willing can place bids, which the broker will try to optimise in the cheapest possible way to benefit itself by performing the query using less money than the offered budget. If a budget is insufficient, a broker refuses to execute the query.

Decentralisation The client-server model assumed till now, causes a concentration on one, or a small number of servers targeted by the clients. To retain service levels, load balancing schemes and fault tolerance algorithms have been developed. However, this concentration problem stimulated research on approaches that try and spread this load over a large number of participants in the network. The class of systems that were devised in this area are typed *Peer-to-Peer* (P2P) systems, where each participant acts both as a client as well as a server. It trades its own resources for those of others. At the heart of each P2P system is an indexing structure to find keys in the distributed network of participants. Examples are [51, 56, 58, 59, 62] where distributed hash tables (DHTs) or other distributed hashing schemes are used to efficiently map data items onto a participant of the system. The expectations of these systems to solve the concentration bottleneck has led to research for database technologies applied to these systems, such as [29, 37, 40]. By the nature of P2P systems, the resulting database systems typically deal with efficiently finding data sources which together are assumed to be the "database". The data sources need not to be relational, or files, but can also be streams or XML documents. Here also mobile and ambient settings get in the picture. Typically, the scene of large numbers of database-empowered sensory systems which learn and exchange information to reach a common goal are the topic [1]. In mobile environments, information is shared through multiple data brokers. Data 'follows' the mobile device, which may be offline for lengthy periods [20].

Stream Systems From a batch-like processing system, recent shift has been towards continuous emission of data via *streams*. Examples can be seen in the stock market "tick" data and various logging applications that generate information about changes or conditions measured by sensors. With the latter becoming economical attractive, new floods of continuous data needs to be processed. While these sensory streams need a fair bit of routing and aggregation in an energy preserving manner, database systems have been adapted to perform those aggregations, or to help execution of formulated queries in an SQL dialect, such as in TinyDB [49]. Monitoring applications that handle streams, may get overloaded by a peak of observation data. In such case it is acceptable for the monitoring application to drop or join observations, reducing the precision in favour of remaining a responsive system. Such operations are in general against the design principles of traditional database systems that try to handle everything, fully correct and in the same order.

2.2 Related Literature

Autonomy, evolution and decentralisation are well known terms in database literature. We briefly discuss the known work on each of these topics.

Autonomy In the research area of federated databases, the notion of autonomous components is rather common [34]. The components that manage the data are often willing to share, as long as they retain to be in control. The autonomy in federated systems stems mainly from the heterogeneity of the participating components. Because of their intrinsic differences it is hard, if not impossible, to leverage precise control on their behaviour. But also the sociological aspect of ownership of the database components plays a role in the setting of federated databases. Even though for instance universities share their databases with others, they prefer to remain in control of their own.

In [71] three types of autonomy are distinguished: design autonomy (D-autonomy), communication autonomy (C-autonomy) and execution autonomy (E-autonomy). *Design autonomy* refers to the ability of a component in a system to choose its own design. Types mentioned for D-autonomy are design selections of the data being managed (universe of discourse), how to represent this data e.g. using which schema, in what way transaction semantics are defined for the local constraints and on what hardware or with which software the database runs. Of course D-autonomy also allows to adopt new approaches on the aforementioned areas whenever that is considered to be beneficial from the component's point of view. This type of autonomy is typically seen in the federated databases case, and used in e.g. [22]. *Communication autonomy* focusses on the ability to decide with who, when and what the component communicates. C-autonomy involves the freedom to decide at any moment whether to communicate, and if so, to some or all other components. This also includes the freedom to be selective on requests itself. A component may for example refuse to execute a query for a given host, but accept the same query when issued from another host. C-autonomy could be a requirement for weakly connected systems, such as mobile or ambient systems [8] where components are expected to be connected to the network for undefined periods at undefined times. For reliable communication in these systems between multiple C-autonomy based components, a non-C-autonomy component has to be used as mediator for their communication. Lastly, *execution autonomy* deals with the freedom of a component to accept or refuse the execution of a request. This is based on the amount of efforts it takes to execute the request. E-autonomy allows a component to refuse to execute queries that exceed for instance a certain time limit. But also the effect of the request on the system, such as congestion or deadlock situations, may be reasons for an E-autonomous system to reject a request. E-autonomy can even result in requirements for other components to be involved in the execution of a request, if the executing component considers this to be necessary.

To conclude, an autonomous system in literature is roughly referred to as a system reluctant to share sensitive or critical information that limits its cooperation with others. A compatible vision of autonomy can be found in e.g. [14]: "*autonomous*, that is, they cooperate to only a limited extent, and do not expose sensitive or critical information to each other".

Evolution In the context of databases, evolution is focused on the contents and schema of a database. This is seen as within the local database, and resolution of problems related to either the changes of the schema, or how to make those changes to the schema or object classes in an object oriented database. Causes that make evolution of this kind necessary include migration of different systems into one, changes of rules, changes in applications and their database requirements and new needs arising from new technologies such as the internet.

Schema evolution typically involves three issues: physical evolution, logical evolution and continuity during the evolution process. In the physical schema evolution, typically the tuning parameters of the database are changed to improve its performance. Here, attributes like buffer sizes, storage types and the number of worker processes are to be tuned to the application needs. Research in this area has focused on mainly automating the database such that it can tune itself. As the amount of tuning knobs continues to grow the perfect combination gets harder to find. Typical example of such self tuning approach is [13] where indexes on the data are automatically chosen based on a workload, as to find a good trade-off between storage overhead and gained performance.

Obviously, many tuning operations that are at the heart of the database system, require it to be taken off-line. During this downtime, the database cannot serve any requests, hence it is important to minimise this downtime.

However, it would be even better if the system was not to be brought down for the tuning operations. This requires changes to the way database systems are built, such that from static structures, dynamic structures are made and for instance indexes can be built while the system remains fully operational, such as for instance in [54].

On the logical side of schema evolution, problems regarding changes in the (relational) database schema are dealt with. While *views* on the data can help to make a schema available in another form, they are only of limited help as typically updates to views over tables are difficult and mostly not allowed by the underlying database. The only option left is to evolve the schema through complex transformations. Schema evolution tools try to assist on these transformations, such as [6]. This assistance is not trivial in any case, as semantic meanings in schemas often require the human in the loop.

Decentralisation In literature, decentralisation is an often used term that comes close to parallelism. Particularly P2P systems refer to decentralisation, because they aim to avoid a central dependency, whereas parallelism does not per sé. In [28] a couple of reasons for using decentralised systems are identified. As noted before at the discussion of autonomy, most organisations want to have control over their own systems. In the light of a distributed system, this means a central controlling entity conflicts with the organisational needs. Hence, a decentralised solution forms a good match. But also heterogeneous systems that are joined together, require their own control. The technical benefits of decentralised systems come close to those of parallel systems. The capacity can grow beyond that of a single system, response times can be improved when the data is placed nearby and availability increases when the systems are geographically widespread.

That the aforementioned benefits can be really attractive, has been proven by the success of P2P systems. Even though their use is mostly for a legally questionable data transfer, fact remains that the current semantics-free, requests for objects by identifier (typically a hash) are quite successful, even though this method is quite limited. P2P systems as such in their current form are quite ineffective to retrieve for instance only the abstract of a given document. Instead the entire document is the granularity of the objects. Data relationships, which are so well understood and supported in database systems, are in particular lacking in P2P systems [29].

2.3 Armada

In the previously described setting of database systems, this thesis explores the path towards evolution, autonomy and decentralisation. Our exploration deals with distribution of shared nothing systems, interconnected by fast networks, as common nowadays. Unlike P2P systems, in Armada we take the database schema and its possible constraints as starting point in our exploration. Distribution and localisation are built on top of the traditional schema approach.

Where the autonomy of a system typically refers to the willingness to share and reveal data to others, in this thesis we lift autonomy to the level where individual systems are the initiators for distribution in the cluster. Due to our schema-centric approach, the systems hosting the data have the required information to do well controlled data distribution. Due to our approach, they also have the required autonomy to perform the distribution using that data. The resulting way of data placement is what we refer to as evolution. In contrast to the known evolution that refers to the data local to a single system, our evolution spans over the entire cluster of participating systems. Hence, a decentralised administration is implied by the autonomous systems that the system comprises of. The essence of decentralisation within Armada is in avoidance of hot-spots in the cluster. Autonomy and decentralisation go hand in hand for this objective.

Our notion of autonomy and evolution are reflected in the first research question of this thesis. In particular the high level of autonomy is articulated by means of *site local decisions*. Our notion of evolution results in the dynamic growth factor, that addresses the entire cluster, instead of a single system. We answer this first research question with a model that encompasses our notion about autonomy and evolution.

With autonomy in place, those who interface with the cluster encounter changes, in particular the clients issuing queries. Because mediators use centralised information about the cluster, their approach does not work in the previously sketched setup of Armada. The DHTs from P2P systems are well suited to locate objects, but they lack support for the schema based approach chosen. For the second research question we therefore explore a stepwise query resolution approach, where clients navigate through the cluster by itself following schema information.

Evolution in databases is expected to become an automatic self-tuning component. However, the growth of a system is not just a matter of making a performance decision based on a workload. It involves an extension to the system that need not to be limiting for following growth operations. In fact, the system needs to be able to keep on evolving all the time, to adjust to changing requirements. This is the area of the third research question.

Summary

Database systems have become an important part of our *world*. Over the years they developed to mature centre pieces in all systems that rely on data storage and retrieval. Being corner stones, database systems have to endure and cope with high workloads and requirements. This lead to the distribution of the database over multiple systems, a trend that continues to become more popular as networks improve. Many aspects of distributed databases, such as parallelism, data placement and replication have been researched to unleash its full potential. Along the way, new visions on the traditional *base* of data have been developed as well, such as wide decentralisation and streaming systems.

With Armada, autonomy of a database is pushed to the level where the local database is the initiator for distribution, based on local conditions. This form of decentralised distribution results in evolution of the entire cluster, which adapts to the workload right there where it is necessary, without a central controlling component.

3 The Armada Model

3.1 Introduction

The *Armada Invencible* is the well known Spanish "invincible" fleet of late 16th century. Full of glory, this grand *armed* fleet wrote impressive history, and yet today its reputation is its invincibility. However, these promising words are not meant to drive up the expectations for the work we present here. In fact, the invincible Armada faced some bitter moments of reality, when it was defeated by Dutch and British ships in the battle of Gravelines. Again, we do not intend to anticipate on any expectations here. Instead, the model that we present in this chapter is named after a fleet of ships. The name *Armada* is used because it is internationally well understood to be such fleet. The discovery of several parallels between Dutch maritime history and our modern database research has also contributed to the title of this thesis. The contents of this chapter is based on the paper "Armada: a Reference Model for an Evolving Database System" [30].

Autonomy The foundation of Armada, is the power of autonomy. Like in a fleet, where each ship has its own autonomous captain. However, even though each captain is independent, efforts are made such that all ships together act as a group, with the same target. The Armada model defines a way to allow autonomous sites to work together as a group towards the same target: serving a distributed database.

Decentralisation Traditional distributed approaches [68] are designed with a strong emphasis on data availability and maintainability. However, these approaches mostly rely on vulnerable centralised techniques. Whenever the central server becomes unavailable (or worse, demonstrates performance problems), the system at large may come to a halt. Also, scaling is often limited by the capacity of the central server, which eventually becomes the bottleneck of the whole system.

This problem has been recognised and targeted by P2P systems. Many distributed hash table (DHT) approaches have been proposed [62, 59, 51]. They form the base for structured overlay networks on top of which P2P database systems are built, such as [37, 29, 40]. DHTs provide associative key-based lookups of data, in the form of a single value, row or block of (cached) data depending on the system's purpose. To find the data, the corresponding key for that data has to be somehow known.

In P2P systems, nodes frequently join and leave the network. This property inherently stems from the environment they are situated in. File sharing between millions of (unknown) people on the Internet simply introduces different time zones, unreliable connections and people unwilling to share (any further). Also, the files available in the network depend on what individual people like to share with others. These — mostly social — aspects of availability and location of data are reflected in the general structure of many P2P systems. They provide an efficient search for data in the network, if it is *currently* available. Data that is currently not in the network, does not exist. Only through stale pointers, data that once existed can be found, but in general P2P networks aim at quickly removing such stale pointers. Hence, there is no notion of a *data space* in these networks that allows determining whether an answer can exist, or does not exist.

Evolution The schema based approach towards distribution as explored in the Armada model delivers a solution for both problems. The model starts from a complete relation and breaks it up into pieces. In this process, it keeps a lineage-based administration of the actions taken. The data within each piece is characterised by a decision function about its permissible content. Moreover, each piece can be recursively broken up further using a refinement relation of the decision function. The pieces naturally map on the data units used in distribution, and the functions are the navigational handles through the ensemble of autonomous nodes.

Armada is a model designed to facilitate the use of both replication and fragmentation. It supports administration of operations for both retrieval and evolution of data with a self-tuning flavour: due to the flexibility of the model, new systems can participate when necessary, old ones can leave, and the actual number of systems or location of data is hidden from users of the system. The

Armada administration allows for localisation of data without need for a central entity that becomes a bottleneck and hot-spot in busy systems.

3.2 Innovations

The Armada model innovates on three key areas, which characterise the capabilities of the model and form the key components towards autonomy, decentralisation and evolution.

Function-based Distribution Control Armada uses general functions to describe the content of a piece. These functions are arbitrary expressions and can be freely chosen on a piece-by-piece basis. They enable an Armada instance to adapt easily to the (local) data distribution or workload characteristics, as part of its *autonomy*. This is in contrast to DHT systems where a single global decision function, e.g. a hash function, is used to control data locality. The Armada model delivers more flexibility by this freedom of choice for functions.

Traditional systems can, in theory, deliver the same flexibility using substantial human efforts, due to the burden imposed by static configurations. The dynamic aspect of Armada aims at adapting on demand by creating pieces with carefully crafted functions to cater for the situation at hand.

Incremental Query Evaluation The Armada query evaluation scheme brings the pieces back into the original relation incrementally, like reconstruction of a jig-saw puzzle. This means that the data is available in the system as ordinary relation fragments. The decision functions allow precise localisation on their logical whereabouts. Even if the piece itself is currently not available in the system, its position in the query result sequence can be derived from its function. This contrasts with DHT systems where this is not known and the only conclusion to be made is that a value that cannot be found is *currently* not available. Functions in Armada describe the data space of the relation. If a value being addressed is not covered by any function, the value cannot possibly exist in any of the pieces. Like in traditional systems, for a query in Armada, it is a priori known what pieces should be inspected before the complete answer has been retrieved. Even if those pieces are temporarily not available. Hence, data that is currently not available, remains to be known to the system via the function administration of the Armada model.

Active Client Participation One can consider traditional centralised approaches to to fail to distribute the metadata required for localisation over the system. The other end of the spectrum is where no central server exists, and instead all individual systems contain the full metadata regarding the data whereabouts. In both situations, the entire system remains a monolithic cluster as long as traditional approaches to transactions and query execution are retained.

For a distributed system to benefit from its infrastructure, a more relaxed transactional setting is required. In such setting for instance locks do not cross a system boundary, hence excluding distributed locks. Armada assumes, like other distributed systems, that the system is used in a loosely coupled setting. Local consistency in the data pieces is required, but global consistency in the system as a whole is left to the client to solve. However, in this thesis we do not focus on the implementation of transactions by clients. With partially distributed metadata and a relaxed transactional setting, the Armada model enables full *decentralised* query execution.

This puts an emphasis on the client's role in the Armada model. In particular, Armada assumes that a client becomes an *active* participant. Active implies that the client plays an important role in steering query resolutions. Instead of relying on the server to construct a complete query result, the client expects a server to answer in portions whenever possible. Not only are partial results returned to the client, but also directions on where and how to get the remaining parts. Here the client is offered some opportunities to influence the execution of large queries, as well as responsibilities to e.g. maintain global consistency.

3.3 The Armada Model

Classical designs for distributed databases, require a centralised server that holds all metadata describing the whereabouts of the available data. Due to the centrality of such server, it can easily become the bottleneck of the entire system. The central server is accessed to lookup or update metadata for both operations that query or update the actual data. More importantly the metadata has to reflect changes in the structure of the system, such as addition or removal of nodes, or a reorganisation of the data for load balancing purposes. It creates a bottleneck that limits the overall performance and scalability of such systems.

Bottleneck An obvious solution to this bottleneck problem is full replication of the metadata over all participating systems. Such designs have to rely on the consistency of the replicated metadata, and hence, each structural change requires a synchronised update on all nodes in the system. However, because all metadata is available locally, data operations are cheaper, at the expense of significantly higher prices for structural changes. While the latter are often infrequent these high costs may be acceptable, but they imply all participating systems to be available and willing to perform the metadata update, with no other update running at the same time. These additional constraints prohibit efficient dynamic changes of the data distribution.

Metadata With the Armada model, we aim at finding a balance between these two extremes. On the one hand, Armada does not come with a centralised server, and thus avoids the bottleneck of metadata lookups. On the other hand, Armada does not require to replicate all metadata on all nodes. Instead, Armada finds a compromise by replicating metadata partially only, and being able to cope with incomplete or stale metadata. Obviously, each node holds its own local metadata, e.g. schema information about the portion of the database stored, and keeps it up-to-date. In addition, it holds some remote metadata, such as information from nodes in its vicinity. To limit maintenance overhead, the idea is to limit remote updates of metadata to those nodes that exchange data due to structural updates. Thus, remote metadata is not necessarily kept up-to-date at all times. Rather, an Armada-node assumes that its remote metadata is an approximation or a past snapshot of the situation of a remote node.

Armada The inspiration for our novel reference model comes from the Armada analogy. An Armada is a fleet of ships, that forms a unity although each ship has a captain who is sovereign. The *Armada model* reflects this property in a minimal set of relations between the captains of the ships. Each ship has cargo (*data*) stored in barrels (*boxes*) that are addressed by cargo documents (*trails*) kept by the captain. A captain can repackage the cargo on his ship, and/or hand over (parts of) his cargo to one or more other ships in the Armada (*cloning*, *chunking*). Repackaging may also occur if barrels are empty or only partially used, such that multiple barrels are put in one (*combining*). The cargo documents describe the content of each barrel as well as the lineage of the respective cargo. A captain keeps one cargo document for each barrel he has aboard his ship. When handing over cargo to other ships, the respective cargo documents are duplicated; the original copy stays with the captain on the old ship and the other one is attached to the barrel on the new ship. Thus, not only does each captain know what cargo his ship currently carries, but also where he sent the cargo that he once had aboard, and where any cargo he ever transported came from. In fact, the cargo documents kept on each ship provide sufficient information to allow the captain to locate any cargo item in the whole Armada [1].

When barrels are transferred to other ships, the captain administrates to who the barrels were transferred in the now obsolete cargo documents. To be able to track down a barrel, copies of the old cargo documents are attached to the cargo documents of the new barrels, and vice-versa. We use the analogy of a real Armada in our world of database servers (*ships*) and apply some of their properties to them.

3.3.1 Notation, Terms and Definitions

We informally introduce the term *(data) box* to refer to the portion of the *data* that is hosted at a *site*. We assume that the content of a box can be described by an arbitrary function g. The actual specification of such function is left to the instantiation of a specific Armada system. In the course of this section, we provide some constraints for such functions. Chapter 4 discusses these functions in more detail and give some simple examples.

Further, we use the term *structural operations* to refer to operations that create and modify the data distribution across sites, i.e., operations that replicate, (re-)fragment or merge portions of the data. Data boxes form the entities that these structural operations operate on.

DEF. 1 *Be $B'_i, B'_{i+1}, \ldots, B'_{i+n}$ existing boxes in an Armada system with functions $g'_i, g'_{i+1}, \ldots, g'_{i+n}$ describing the content of each box. A* structural operation o *operates on one or more boxes $B'_i, B'_{i+1}, \ldots, B'_{i+n}$ and produces one or more new boxes $B_j, B_{j+1}, \ldots, B_{j+m}$ with functions $g_j, g_{j+1}, \ldots, g_{j+m}$ describing the content of these new boxes. A structural operation cannot generate new data, but must not "loose" any data, either. Hence, we require that*

$$g_j \cup g_{j+1} \cup \cdots \cup g_{j+m} = g'_i \cup g'_{i+1} \cup \cdots \cup g'_{i+n} \quad .$$

Inspired by the cargo documents of the Armada analogy, we introduce *lineage steps* and *lineage trails* to store and administer metadata. A *lineage step*

[1]The trail administration for each box is only valid at the time it is created. Afterwards, its references to successors may be outdated. For the site hosting the box this does mean, however, that it can reach the rest of the Armada through the sites it knows as stored in the trails, even though that might not be the most up-to-date state.

captures the logistic information of applying a structural operation to a box:

g, the function that is applied (and hence describes the content of the new box),

S, the site that the new box is shipped to, and

B, the identifier of the new box (for the convenience of later reference).

DEF. 2 *A* lineage step $s = [\, g\ ,\ S\ ]{:}B$ *is a composition that identifies the application of a structural operation, resulting in a new box B on site S with function g describing the content of the new box. The box B′ that s is applied to is identified by the* lineage trail T' *that s is appended to (see below).*

Each box in the Armada is uniquely identified by a *lineage trail* that captures the whole history of its data.

DEF. 3 *A* lineage trail*, or* trail *for short,* $T = s_1.s_2.\cdots.s_l$ *is a sequence of* $l \in \mathbb{N}$ *lineage steps. With* $s_l = [\, g\ ,\ S\ ]{:}B$ *, T identifies box B on site S.*

DEF. 4 *Be* B''*,* B'*, and* B *boxes on sites* S''*,* S'*,* S *with their content described by functions* g''*,* g'*,* g*, respectively. Further be* B''*,* B'*, and* B *identified by the trails* T''*,* $T' = T''.[\, g', S'\,]{:}B'$ *and* $T = T'.[\, g\ ,\ S\ ]{:}B$ *, respectively. We call*

T''	*a* predecessor trail *of box* B',
$s' = [\, g', S'\,]{:}B'$	*the* local step *of box* B',
$T' = T''.s'$	*a* local trail *of box* B',
$s = [\, g\ ,\ S\ ]{:}B$	*a* successor step *of box* B',

and analogously for boxes B'' *and* B.

The metadata maintained and stored for each box consists of *a set* of predecessor trails, *exactly one* local step, and *a (possibly empty) set* of successor steps. The predecessor trails represent the box' heritage. The local step describes the box itself, and the successor steps point to the box' offspring. The predecessor trails and local step are set upon creation of a box, while the successor steps are only set once a box participates in a structural operation.

We assume that a structural operation (logically) removes all the data from its input boxes (transferring it to the newly created boxes), and destroys the input boxes. Only the respective metadata (lineage) is kept. This assumption relieves us from the need to consider different versions of each box, and thus helps to simplify the model. The assumption does not limit the generality of the model. In a practical implementation, this does not necessarily require a (physical) copy of all data with each structural operation. Instead, simply renaming the box can be sufficient.

To simplify the presentation, we omit the set notation whenever a set of trails is empty or contains only one trail. In the first case, we simply omit the empty trails set; in the latter case, we depict the only element as singleton. Thus, the metadata for boxes B'', B' and B of Definition 4 is depicted as follows:

	pre	*loc*	*suc*
$T'' =$	T'''	$.\ [g'', S'']{:}B''\,;$	$[g', S']{:}B'$
$T' =$	T''	$.\ [g', S']{:}B'\,;$	$[g\,, S\,]{:}B$
$T =$	T'	$.\ [g\,, S\,]{:}B\ ;$	

The set of successor steps is empty for all boxes to which no structural operation has been applied yet, i.e., all boxes that physically exist and store data. The set of predecessor trails is empty for one box in an Armada, the *origin*.

DEF. 5 *An Armada instance is born as a single initial box B_o. We call B_o the* origin *of the Armada instance. Obviously, the origin has no predecessor trails. Further, since no structural operation is applied to create the origin, there is no function that describes (restricts) B_o's content. We indicate this by % in the local step of B_o:*

$$T_o = [\,\%, S_o]{:}B_o\,.$$

3.3.2 Structural Operations

To let an Armada evolve from the origin, we consider the following three structural operations.

Replication: the *clone* operation

DEF. 6 *The* clone *operation operates on one box B' with function g' and generates one or more new boxes $B_j, \ldots, B_{j+m}$ that all contain a copy of B''s data. Hence, their functions $r_j, \ldots, r_{j+m}$ are all identical to g'.*

Replicating a data box is the action of copying its content to a new location. We call it the *clone* operation, denoted by function r. Consider the following example of cloning the origin box B_o:

$$\begin{aligned}
T_o &= & [\,\%, S_1]{:}B_o\,; &\left\{\begin{array}{l}[\,r\ , S_1]{:}B_1\\ [\,r\ , S_2]{:}B_2\end{array}\right.\\
T_1 &= [\,\%, S_1]{:}B_o\,. & [\,r\ , S_1]{:}B_1\,; &\\
T_2 &= [\,\%, S_1]{:}B_o\,. & [\,r\ , S_2]{:}B_2\,; &
\end{aligned}$$

In this example, the origin has two successors, B_1 and B_2, which themselves have no successors.

From the origin meta-data we can now observe two trails by reading from left to right: each of the successor steps, followed by the local step and the predecessor trail. The full local trail for the two new boxes (successors of B_o) is also visible.

Following Definition 6 the number of new boxes produced can also be a single one. Strictly, this is no cloning operation any more: since the original box is (logically) destroyed after the cloning, its data is not replicated, but rather moved to a single new location. However, there is no reason to prohibit this in the model.

Although we use different site identifiers for the two new boxes in the above example, it is perfectly sound with the model to produce two (or more) clones of a box on the same site. The question, whether this is reasonable in practice, is not relevant in the context of a reference model.

Fragmentation: the *chunk* operation

DEF. 7 *The* chunk[2] *operation operates on one box B' with function g' and generates one or more new boxes $B_j, \ldots, B_{j+m}$ that all contain a fraction of B''s data. We require that all fractions are disjunct, but no data is lost, i.e., the following must hold for new boxes' functions:*

$$f_j \cup \cdots \cup f_{j+m} = g' \qquad \textit{and} \qquad \forall_{k,l \in \{j,\ldots,j+m\}, k \neq l} : f_k \cap f_l = \emptyset \qquad .$$

Fragmenting data means it gets spread out over multiple boxes. We call this the *chunk* operation, denoted by functions $f, f', f'', \ldots$. Consider the following example of chunking the origin box B_o:

$$\begin{aligned} T_o &= \qquad\qquad [\,\%, S_1]{:}B_o\,;\ \begin{cases} [\,f\ , S_1]{:}B_1 \\ [\,f', S_2]{:}B_2 \end{cases} \\ T_1 &= [\,\%, S_1]{:}B_o \,.\, [\,f\ , S_1]{:}B_1\,; \\ T_2 &= [\,\%, S_1]{:}B_o \,.\, [\,f', S_2]{:}B_2\,; \end{aligned}$$

The origin has been chunked into two parts, using chunk functions f and f'. Like with cloning, in case there is only one result box, a move operation is effectively being executed.

[2]We felt free to 'invent' this verb.

Merging: the *combine* operation

DEF. 8 *The* combine *operation operates on one or more boxes* $B'_i, B'_{i+1}, \ldots, B'_{i+n}$ *with functions* $g'_i, g'_{i+1}, \ldots, g'_{i+n}$*, and produces a single new box B that combines all the data of the input boxes. The produced box' function m spans the domain of* $g'_i \cup g'_{i+1} \cup \cdots \cup g'_{i+n}$.

While cloning and chunking are growing operators, the *combine* operation is a shrink operation. Applying it to a number of boxes merges them into one. However, this operation is not restricted to acting as an inverse-operation to the clone and chunk operations, i.e., re-constructing a previously cloned or chunked box. Our model allows to apply the combine operation to an arbitrary set of boxes. This is depicted in the following example, where a clone (B_4) and a chunk (B_6) are combined into a new box (B_9), creating a duplicate free combination of the inputs' data.

A note on the notation is necessary: for convenience, clarity and space reasons we do not write down the predecessor trails. From now on, we use a reference to them in the form of T_x where possible instead.

$$\begin{aligned}
T_4 = &\quad T_3\,.\,[\,r\ \ , S_1]{:}B_4\,; \quad [\,m, S_1]{:}B_9 \\
T_6 = &\quad T_2\,.\,[\,f'', S_2]{:}B_6\,; \quad [\,m, S_1]{:}B_9 \\
T_9 = &\quad \left.\begin{matrix} T_4 \\ T_6 \end{matrix}\right\}\,.\,[\,m, S_1]{:}B_9\,;
\end{aligned}$$

Again, if there is just one box merged, the result is a semantic move of data.

3.3.3 Lossless Principle

The clone, chunk, and combine functions permit an arbitrary Armada constellation to be constructed. It would even allow for destructive functions, i.e., creating rubbish.

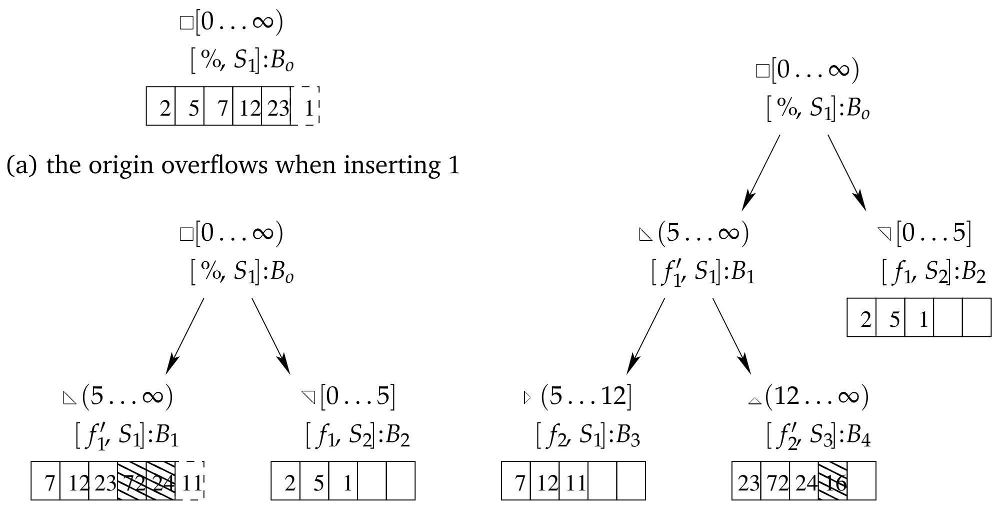
An important class are the lossless constellations. That is, at any point in time it remains possible to combine the boxes on a single site with infinite resources without loss of any box content.

This property is fulfilled for clone operations by definition. For the chunk operations it limits their definition. It precludes aggregations and general (schema-based) data transformations.

3.3.4 An Armada Database

In practice, databases based on the Armada model evolve over time quickly. For many reasons, e.g., resource limits, boxes are the target of *chunk* and *clone* operations. An illustrative example of a database with 5 boxes is shown below.

$$
\begin{aligned}
T_o &= \quad [\,\%,\, S_1]{:}B_o\,; \begin{cases} [\,f_1,\, S_1]{:}B_1 \\ [\,f_1',\, S_2]{:}B_2 \end{cases} \\
T_1 &= \quad T_o\,.\,[\,f_1',\, S_1]{:}B_1\,; \begin{cases} [\,f_2,\, S_1]{:}B_3 \\ [\,f_2',\, S_3]{:}B_4 \end{cases} \\
T_2 &= \quad T_o\,.\,[\,f_1,\, S_2]{:}B_2\,; \\
T_3 &= \quad T_1\,.\,[\,f_2,\, S_1]{:}B_3\,; \\
T_4 &= \quad T_1\,.\,[\,f_2',\, S_3]{:}B_4\,;
\end{aligned}
$$



(a) the origin overflows when inserting 1

(b) box B_1 overflows when inserting 11

(c) the final state of the Armada

Figure 3.1: Sample Armada with 5 boxes.

In this example, we only use fragmentation functions to spread the data in the Armada over 5 boxes. Each box is hosted on a separate site for ease of presentation. The origin box B_o was first chunked into boxes B_1 and B_2. The first of these two children, B_1 is chunked again, resulting in boxes B_3 and B_4. The evolutionary steps are graphically shown in Figure 3.1 using symbols which indicate the coverage of the functions applied in the operations on the boxes. The symbol '□' is used to represent the data at the origin of the Armada, in box B_o. The other symbols; ' ◹ ', ' ◺ ', ' ▷ ' and ' △ ' represent pieces of the origin box. Note that the symbols equally divide the original square symbol. This is of course only a drawing issue, which is not necessarily true for the fragmentation functions being used.

For this example, we describe how the tree from Figure 3.1 is built over time by inserting data into the Armada. In the initial situation, depicted in Figure 3.1a, only B_o exists on site S_1. For the sake of the example, the boxes store simple integer values. Each box has a fixed capacity of 5 of such integers. Normally this capacity is determined by the site that hosts the boxes and the size of the data items, but for the sake of clarity we use these fixed sizes. The data to be inserted in the Armada, in order, is for the example:

$$D = \{2, 5, 7, 12, 23, \quad 1, 72, 24, 11, 16\}$$

Since there only fit five integers in each box, the origin B_o consists of $D(B_o) = \{2, 5, 7, 12, 23\}$ when the next integer, 1, is attempted to be inserted. Since it does not fit, a chunk operation is performed. In our example, we split equally, which results in $D(B_1) = \{2, 5, 1\}$ and $D(B_2) = \{7, 12, 23\}$. The fragmentation function f_1 used here selects the range $[0 \ldots 5]$. The function f_1' selects the complement of f_1: $(5 \ldots \infty)$. Beware, this decision is taken at site S_1 in 'full autonomy', it is not inherent to the algorithm.

In Figure 3.1b, the state of the Armada after the first chunk operation is depicted. As can be seen, the data from the origin box B_o has been moved to boxes B_1 and B_2. Note that the order of the items in the example is maintained, but this is not a restriction of the Armada model. The only restriction on the boxes is that each box only holds data that matches its respective local trail description.

Continuing the insertion of values, now the right box has to be searched. Inserting the values 72 and 24 ends up in box B_1. The origin box B_o is not active any more, and redirects if being consulted. Since it knows the functions of its successors, it can easily tell that both values fit in the $(5 \ldots \infty)$ range of B_1 [3]. Also the next integer, 11, fits in B_1's range, but since the box is full, a chunk operation has to be performed again. The result of this chunk operation is depicted in Figure 3.1c. Again the data values have been equally split over

[3]A more detailed description of how this redirection is decided upon is given in Section 3.3.5.

the two new boxes B_3 and B_4. The last integer to insert, 16, ends up in box B_4 guided by the ranges associated with the active boxes B_2, B_3 and B_4.

From the example it can be easily seen that the different functions r, f and m end up in the trails for the various boxes. For each box *lineage* can be seen in the predecessor trails, which grows every step by extending the lineage information of the box being operated on.

3.3.5 Localisation

A client can query the Armada by sending its query to one of the Armada's sites (S_1, S_2 or S_3). Multiple boxes can be hosted on a single site, hence sites have access to all of the trails that belong to the boxes they host. A query directed to a site, can then be evaluated by the site to see if it can (partially) handle the query. Based on the functions present in the trails, data coverage and query span can be evaluated. As a result of the administration of predecessors and successors in the meta-data, a hint can be given into the right direction if (parts of) the query cannot be handled.

Successful and efficient localisation of the box(es) that potentially hold the requested data is a vital prerequisite to allow query execution on an Armada system. Using the previous example, we now briefly sketch that the lineage trails provide sufficient information to find the responsible box(es) for the requested data.

Note that when clients contact the Armada, they are contacting one (or more) of its sites that host boxes, not the boxes themselves. The example from Figure 3.1c describes 5 boxes that are in fact hosted on 3 sites, S_1, S_2 and S_3.

Point Query Suppose a client c has a query which is answered by △, say 42. c can now contact any of the sites from the Armada. Any site that cannot handle the request by c, redirects c to the site that it knows has more specific information. The simplest case is when c connects directly to S_3. On S_3, only trail T_4 is available. This trail defines the box responsible for the data fragment $(12 \ldots \infty)$. There are no successors for B_4 available, meaning B_4 is active. Trail T_4 tells that the query for △ can be answered. In our example this means that S_3 can tell c that there is no 42 in the Armada.

In case c connects to S_1, S_1 has three trails at its disposal: T_o, T_1 and T_3, where T_3 is the most "specific" trail. Evaluating from that trail, c's query cannot be answered, hence a redirect to the predecessor box has to be made. (There are no successors to consider for T_3.) Since the predecessor box T_1 is on the same

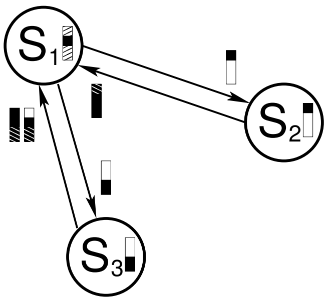


Figure 3.2: 3 sites in an Armada and what they know of each other.

site, the redirection can be done internally, resulting in no client redirection. Evaluating T_1, c's query can be answered, but since box B_1 is no longer active, it must be answered by one of its successors. In this case by successor T_4, which is located on site S_3. Hence, a redirect to c for site S_3 is sent. As obvious from the previous case, at S_3, c retrieves the answer to its query.

Finally, c can decide to connect to S_2. At S_2, the trail T_2 is available. This trail does not cover the query △ , so neither would its successors do, if any. Hence, a redirect to the predecessor box is sent. This box, the origin B_o, is located on S_1. Since S_1 does not (have to) know that c was redirected for box B_o, it just evaluates c's query like it did in the case above, with the same result.

Range Query So far we only considered a query which was fully contained in a single box: the lookup of the value 42. Instead of this point query, a range query could be issued by c, that possibly spans multiple boxes. Consider query △ which describes a range $[10 \ldots 20]$. Like in the previous cases described, client c ends up at sites S_1 and S_3. Both sites are able to return a *partial answer* to the query and an additional redirect in order to get the remainder of the answer. Here, the client has to deal with the data being spread over two sites.

It must be noted that for this example we chose to have three different physical sites. This is merely for explanatory purposes. It is very well possible for every box to be on its own site, or for all boxes to be on the same site. There are no inherent restrictions in the Armada model as to where boxes are hosted.

Relations The lineage trails from the Armada model, resemble the relations between the sites and the function coverages. Every step from the predecessor and successor trails at a site contains a site and a function. The example Armada from Figure 3.1c, can be depicted as in Figure 3.2. In the latter figure, only sites are shown, and their relations to other sites, represented by arrows. Next to the relations between the sites, Figure 3.2 also depicts (chunk) functions as a range in a finite space, for ease of understanding. The shaded area at S_1 represents the range covered by functions of no longer active boxes, the black area the current range covered. The outward arrows from S_1 show the function coverage at the head of the two arrows. As can be seen from the figure, the black areas in the ranges of S_1 and the outward arrows together cover the full range. This is due to each operation being performed with S_1 involved. That this is not the case for S_2 and S_3 can be seen from their arrows back to S_1. From the trail available at S_2, T_2, it is only known what the local function is, f_1, and what the predecessor function is, %. As a result, S_2 does not know about f_1', hence when a request falls outside of its own function coverage, all it knows is that its predecessor (the origin) contains “all” data. Of course it never redirects for the data that is locally covered, as indicated by the striped area in the figure. The same holds for S_3, where T_4 is available. It has two steps in the predecessor trail up to the origin. Next to the origin itself, like for S_2 the step from T_1 contains the function f_1'. In a redirect, the site can use both to determine which site comes closest to what data is requested, which in the example happens to be the same site, but not necessarily has to be in larger Armadas. In Chapter 6 redirection strategies based on the functions available are discussed in more detail.

For every site, the whole Armada can be constructed by combining the functions from all outgoing arrows with the local function. This is easily deduced from the range representation in the figure, since combining all the ranges result in full coverage of the entire space. This combination is used in query resolving and composed of the local and remote functions as known from the local, predecessor and successor trails. When a query is executed at a site, it is executed using this combination. Since each site has its own combination of functions which represents the same — the whole Armada — actual execution may differ from site to site, but the final outcome is the same.

3.4 Lineage Wrecks

Eventually, each Armada that has a form of continuous growth becomes large. In principle, such large tree is no problem, however while data can be moved

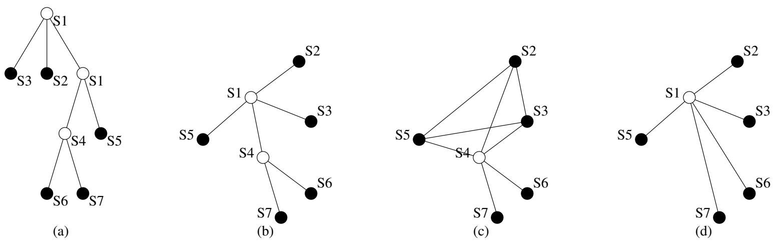


Figure 3.3: The same Armada tree, in (a) represented as lineage tree and in (b) represented as association tree. (c) represents the association tree after S_1 has been removed, (d) after S_4 has been removed.

$$
\begin{array}{rl}
T_o = & [\%, S_1]{:}B_o\,; \begin{cases} [f^1, S_1]{:}B_1 \\ [f^2, S_2]{:}B_2 \\ [f^3, S_3]{:}B_3 \end{cases} \\
T_1 = & [\%, S_1]{:}B_o\,.\,[f^1, S_1]{:}B_1\,; \begin{cases} [g^1, S_4]{:}B_4 \\ [g^2, S_5]{:}B_5 \end{cases} \\
T_2 = & [\%, S_1]{:}B_o\,.\,[f^2, S_2]{:}B_2\,; \\
T_3 = & [\%, S_1]{:}B_o\,.\,[f^3, S_2]{:}B_3\,; \\
T_4 = & [\%, S_1]{:}B_o\,.\,[f^1, S_1]{:}B_1\,.\,[g^1, S_4]{:}B_4\,; \begin{cases} [h^1, S_6]{:}B_6 \\ [h^2, S_7]{:}B_7 \end{cases} \\
T_5 = & [\%, S_1]{:}B_o\,.\,[f^1, S_1]{:}B_1\,.\,[g^2, S_5]{:}B_5\,; \\
T_6 = & [\%, S_1]{:}B_o\,.\,[f^1, S_1]{:}B_1\,.\,[g^1, S_4]{:}B_4\,.\,[h^1, S_6]{:}B_6\,; \\
T_7 = & [\%, S_1]{:}B_o\,.\,[f^1, S_1]{:}B_1\,.\,[g^1, S_4]{:}B_4\,.\,[h^2, S_7]{:}B_7\,;
\end{array}
$$

Figure 3.4: The Lineage Trails for the Armada tree depicted in Figure 3.3a.

to release sites from their data load, the sites themselves cannot be removed. The lineage trails in the tree in this respect “claim” each site to avoid a gap in the routing scheme, thereby preventing their removal.

The Armada model defines lineage trails to be immutable, hence simply “updating” their contents is impossible. Instead of reconsidering this immutability, we simply define additions to the lineage administration, which are in line with the original model. These additions take the form of an extra trail, which we refer to as *jump trail*. Jump trails are trails not pointing to direct predecessors or successors, but to any other box in the system. By carefully adding such jump trails, boxes in the lineage can be bypassed, thereby rendering them obsolete. Note that trails are never removed, hence references to sites remain to exist for the life-time of the Armada.

In Figure 3.3a and 3.4 an example Armada is depicted by its lineage tree and trails respectively. In addition to the lineage tree, an association tree is shown for the same Armada in Figure 3.3b. The example describes a chunk-only situation, where S_1 and S_4 became empty after they were chunked. This is in contrast to previous examples where one part of the chunk operation remains on the original site. Because S_1 and S_4 are empty now, they only send redirects to their offspring when consulted. In the situation of the example, both sites have no other function than redirecting, as they store no data.

The sites S_1 and S_4 are due to their limited use good candidates to be removed from the Armada. In fact, a planned removal could be the reason for chunking to two new sites. While a site cannot be "removed" from the Armada, it can become unavailable with no intentions to return. While the site remains to be referenced in the trails of the Armada, the information stored at the site remains necessary for the Armada as a whole. Hence, a ship that becomes a *wreck* — a site that purposely is removed from the Armada — hands over its lineage information to another site in the Armada. The most obvious candidates for this are the predecessor sites for all hosted boxes. In the example, sites are not reused, hence there is only one predecessor site for each site, except the origin.

Predecessor Propagation For site S_4 to become permanently unavailable, it has to transfer its unique knowledge to its predecessor. The unique knowledge is the case of S_4 the successor steps to S_6 and S_7, as specifically can be seen in Figure 3.3b. Without trails pointing to those sites in the Armada, others cannot reach them, while S_6 and S_7 themselves could still reach the rest through S_1, which they inherited in their predecessor trails. The jump trails to add to S_1 resemble T_4. With those specific trails, S_1 needs not to redirect to S_4 any more, since with the jump trails added it has access to more specific trails that supersede the original one for S_4, as depicted in Figure 3.3d.

Successor Propagation In case of S_1, making the site unavailable is not as simple as described before, since there is no predecessor to hand over the unique knowledge to. A conceptually simple solution would be to assign the trails of S_1 to another node in the system. This node would become the origin, and the whole system could continue to work as before, using the new origin. However, even though this sounds trivial and simple, it means *all* lineage trails in the entire Armada need to be changed in order to reflect this change, since each trail contains the full lineage up to the origin. Apart from being a very expensive operation, this can only be done by adding extra trails, which in this case are not more specific. For both reasons, this solution is deemed not to be a viable solution. It may be evident that it cannot be overcome that lineage trails point to a no longer existing node, e.g. S_1.

Instead, in this special case, the unique knowledge has to be pushed down to the direct successors of S_1. Again, the idea here is to make S_1 obsolete by having more specific information available. Figure 3.3c depicts the situation where the trails from S_1 have been pushed down to all successors. If S_6 would send a redirect to S_1, this redirect points to a wreck. When this is encountered, a search for surrounding sites using the lineage trails is done, in this case ending up at S_4. The latter site is then able to redirect to the appropriate site to continue the search.

3.5 Related Research

Close to Armada's objectives is Mariposa [65]. This system which we briefly mentioned in Chapter 2 aims next to its economical decisions for a distributed setting based on fragments of data among autonomous systems. Unlike the envisaged client interaction in Armada, Mariposa passes queries or data on to other sites it knows on behalf of the actual client, resulting in a chain of dependent systems representing the economical broker structure. Further on, location of fragments is not really specified and forms part of the bidding process, whereas Armada has this embedded in its lineage trails. The lineage information in Mariposa is used mainly for merging back previously split fragments. Armada on the other hand, allows merging of any two or more boxes. While Armada does not explicitly deal with data mobility, heterogeneous host capabilities and a simple language that controls actions done on the data, it does not outlaw their use. In fact, the Mariposa rule system defines action routines that map on the clone, chunk and combine operations, and the related fragmentation functions.

Stream Databases In recent years, two research trends in distributed databases have emerged. The first are sensor network databases are characterised by a large number of resource limited receptors at the edge of a network to collect mission critical data. Prototypical building blocks are small 'Motes', a single-board-computer (SBC) equipped with limited memory, limited network capabilities and limited energy, glued together to form a distributed information system to feed the upstream applications. On each site, we find one or more sensors and an embedded SQL database engine for storage management and query processing [48, 21, 5, 2]. However, their underlying architectures ignore autonomy as we aim for with Armada. In essence, they are built from functionally scaled-down versions of relational database systems.

P2P Systems Second, Peer-to-Peer (P2P) systems have gained a lot of interest. A comparison of Armada with such P2P systems is inevitable, since both systems are decentralised using highly autonomous participating nodes. The focus of P2P systems is efficient query routing and localisation [56, 52], yielding in a *routing-centric* view. Armada differentiates from this approach in having a *data centric* view: the data, in terms of boxes, filled with relations are aimed at evolutionary growth starting from a single node. This different point of view yields in some elementary differences between Armada and P2P systems. These differences are all related to the way data is distributed over the system. P2P techniques assume the data is already in place and numerous, usually in the form of files. In general the placement is not bound to any rule, and usually simply on the machine that provides the data on the network. Replication of that data is a side-effect of other machines that copy the data and make it available afterwards. P2P systems in general do not make any efforts to manage the data that is in the system. Instead, they focus on how to find this data in the network, using a key-based approach, where each data item is assigned a key used for lookup. Here, efforts are made to have this lookup structure being fast and resistant to failing nodes.

Unlike P2P systems, Armada has functions that define how data is split over a number of boxes, which allow for concise localisation of data. With Armada, the focus is on the data being stored. Using a schema-based approach, instead of dividing the key-space Armada divides the data-space over multiple machines. Here, the data is split into parts based on a machine local function, that suits best for the scenario at hand. Load, capacity or redundancy problems on a local machine are the trigger within Armada to split the data to another node. Here it must be stressed that the split function, is local to the machine unlike in P2P systems which use e.g. a predefined hash-function for the whole system. The latter of course, only deals with the key-values, not the data itself. As a result, Armada nodes are able to resolve local overflow situations by taking an autonomous decision to split their data and move a part to another node. The index to find the data, hence is not based on some key but the value itself, the data. Because keys in a DHT system are generated by a hash-function, data properties are lost in the key representation. This is not a problem for point queries which look for a certain (upfront known) key, but it does complicate efficient range queries, that target the data values. Querying key ranges, which is not comparable to data value ranges, needs help to simulate range support such as in the Distributed Segment Tree [74].

A P2P system eventually never searches in the data value space. Instead, it solely looks at key values. In this regard, Armada is more suitable for traditional database use, as those databases work with the data values as well. P2P database systems, such as PIER [37] and PeerDB [55], hence only concatenate database systems or data sources. They simply lookup the appropriate sources and apply the queries on those. A fragmentation of a single data source, in a distributed manner such as in Armada, is not being dealt with. P2P systems that would assign a key to each data item, e.g. a database tuple, assume that a query in such system exactly knows what it wants to retrieve — which defeats purpose of the query.

In more detail, the objectives of PIER [37], a P2P database system, differ from Armada in what is provided by the *convergence* — eventual consistency as also found in epidemic-based systems [70] — of the Armada model. PIER focuses on massive distribution to validate scalability and distribution. Traditional ACID properties are relaxed because that is a necessity [26], but unlike PIER, Armada maintains the global schema requirement for the data. Another P2P related structure, BATON [38], is a tree shaped P2P overlay network. Whereas BATON is a balanced binary tree, Armada uses generic (heterogeneous) functions and needs not necessarily to be balanced. In fact, node relocations in the tree are not supported in Armada, because the tree is built out of the lineage relation between the nodes, that cannot possibly change in the same way that history is never rewritten.

SDDS Scalable distributed data structures (SDDS), a predecessor of P2P systems, use globally known, but locally adaptive partitioning functions [45, 42]. Also the client behaviour in SDDS implementations bears some similarity with the Armada approach. They manage a cache with metadata to direct data lookups. The main difference with the Armada vision is its level of abstraction. SDDS solutions are focused on single key-based retrieval. In our model, we extend the scope to the complete functionality of a database system. Furthermore, the lineage trails capture the complete history of a box, something not considered in an SDDS. It maintains the latest, locally consistent distribution status.

Self-management Over their life span, database systems experience a continuous change (usually growth) of the amount of data stored. Likewise, usage patterns and workloads keep on changing. For example, more recent data is often accessed more frequently than older data, creating a "continuously moving access hotspot". Classical distributed database architectures hardly provide any means to adapt to these changes automatically. Evolution techniques are mainly based on local conditions. Rather, increasing the system's capacity (by adding additional nodes) and re-distributing the data to balance the load are measures that have to be initiated and executed by some human DBA [68]. Additionally, client/server settings form the base of dealing with the work, where servers perform the entire job of query execution as "service" for the client. The result is a reduced autonomy of servers from an implied work point of view. Servers have to go through the full execution, instead of only the part they are responsible for.

The area of self-managing and self-tuning databases limits itself by only advising the DBA [57, 75] or only dealing with indices and materialised views [3] — the metadata. Combinations of replication and fragmentation are not supported, and only on the whole table data, where fragmentation is only horizontally applied. Armada, on the other hand, can be considered a self-adaptive model to meet the environment requirements and reconfigure when they change. A compatible vision can be found in distributed systems, where decentralisation is the key as well [70].

Summary

The Armada model is a schema based solution to distribution. The control over distribution parameters is set by functions that divide the data into (smaller) pieces. The function can be freely chosen by the site that performs the operation, leading to ultimate autonomy for that site. Not only can it decide when, but it can also decide how to perform the operation, thereby supporting incremental scalability.

Each function that is applied is recorded in trails that track how pieces of data in the Armada have evolved. These trails are stored decentralised in such a way that localisation of the pieces is possible from any site in the Armada. Sites that are unavailable are not "forgotten", but instead remain present in the trails, leading to a consistent image; data does not suddenly appear or disappear, it is or was known to be missing instead.

To retain the autonomy of the sites in an Armada, sites do not perform work for others. Instead they send redirects or refuse to work. This requires an interaction shift from passive to active, where the client is expecting to follow redirects and deal with the structure of the Armada to resolve a query.

A big contrast to other systems is that Armada uses a data centric view on distribution and the arbitrary functions that follow out of that. The high value for autonomy in the system, to make it self managing, is not to be found in most other systems. The self management for the cluster as a whole that Armada aims for, goes beyond the single (sub-)system level and enables a continuously evolving system.

4 Automated Operations

4.1 Introduction

The Armada model tracks the evolutionary path of data within a cluster of machines. Where data is placed is precisely administered through lineage trails. Important in those trails is the use of functions that specify what part of the entire data is affected by the operation. A large part of the flexibility of Armada is due to the freedom of choice in what function to use for each operation. A part of Armada is the autonomy of sites to initiate operations to take place by themselves. This yields in the question how the functions can be applied without human intervention, such that the Armada can evolve itself where necessary. This kind of evolution where the system adapts itself is a form of self-adaptation from the self-managing area.

Already back in 1989, Self-*Something* aware databases were proposed, such as Cactis [36], a self-adaptive, concurrent implementation of an object-oriented database management system. The self-adaptiveness of Cactis is in the dynamic change of the physical organisation and order of update algorithms to reduce disk access. Many more self-* were to follow, all focused on optimising a local condition automatically without human help. Nowadays, self-healing refers to detecting and restarting failed or hung services [17]. SQL Anywhere [10] does database self-management, which means that the buffer pool size of the DBMS is increased and decreased based on feedback from the OS and paging faults. Commercial database management systems take the term self-management as the ability to avoid a performance nightmare. Instead of seeing spikes in the performance of the system, a graceful degradation is achieved by e.g. applying control theory, such as in [67].

Recently virtual machines have gained a lot popularity. They impose different behaviour characteristics due to the shared physical hardware. In [15] the inference caused by the virtual machines in a replica setting is discussed. The self-managing nature here is to figure out what queries cause a lot of interference and deal with them either via complete relocation on another replica, or by limiting the resource quotas.

The work done in the self-* area mainly focusses on local tuning operations. Where other sites are involved, those sites are used to solve a local performance issue. In an Armada cluster, the self-adapting nature we are after goes beyond a single system and instead focusses on the entire cluster. What we're looking for is more of the self-organising nature of multi-agent (MA) systems [73] in this respect. MA systems consist of a number of agents that independently try to work on the same target. Here, a maximisation of the *utility* of the whole system is preferred over the *utility* of an individual agent. In other words, one may have to suffer for all others, or the system at large to become better. Each site in an Armada can be seen as an agent, and as such the Armada itself as a MA system. The sites in an Armada work together on serving the work and storage loads of a DBMS. Maximising the efficiency of the whole Armada is achieved by both spreading data for performance, dividing data because of capacity, and replicating data for redundancy.

The first sections in this chapter deal with the operations available in Armada and the functions that can be used for them and their characteristics. The last sections dive into the self-managing aspect of the Armada and how the functions should be operated in such setting.

4.2 Functions and Operations

Distributed databases come either as replicated or fragmented variants. Proposals for combinations of the two are rare and most systems focus on either one of the two. Since replication is more an effort of coordinating machines, usually through an hierarchy [60, 19], not much can be said about the data operations for it. Fragmentation on the other hand is more challenging. Two types of fragmentation are distinguished, horizontal and vertical fragmentation. They refer to how the data relations are cut into pieces. In the case of horizontal fragmentation this means a row-wise cut, resulting in fragments containing full tuple rows. Taking the *union* of all horizontal fragments yields in the original relation. Vertical fragmentation cuts column-wise, with as result separate sets of columns as fragments. Obviously reconstruction here is done using a *join* between the column sets.

At the base of horizontal fragmentation are predicates that describe which tuples belong to which fragment. Traditional literature considers these predicates to be a prerequisite for fragmentation [12]. Predicates are simple attribute conditions, and the process of determining how to fragment is driven by identified predicates based on application usage. The classical example uses the geographical location attribute with a condition that separates two regions of interest. This condition is taken from major application queries, such that the fragmentation is supportive to the system. The idea behind this is that when a fragmentation is chosen that results in consulting all fragments for each query, the fragmentation does not help the system, since networks are slow. This has improved, and possible gains from parallelism contrast this way of thinking.

When more attribute conditions are identified, a *minterm* predicate is used to identify the conjunction of conditions that are relevant. The minterm predicate takes simple predicates either in their normal or negated form, but only in conjuction. This means no combinations can be made which extend each other as in the boolean OR logic. Thus:

$$y = \bigwedge_{p_i \in p} p_i^*$$

with p_i^* being either the normal or negated form of p_i. Further the conditions may not contradict, hence $y \neq false$. Resulting are the *relevant* predicates considering the identified important application queries.

In vertical fragmentation, columns are grouped in sets, which are like in the horizontal case identified by important applications' queries. If an application needs attributes from more than one column set that application does not benefit from the fragmentation as the columns need to be joined again to reconstruct the original relation, again with slow networks in mind. To aid in this a tuple identifier like a primary key or row id needs to be present in each column set. This implicitly adds duplication of data when using vertical fragmentation. However, in today's technological state, the benefits of performing e.g. selections on columns in parallel can be quite substantial.

Dropping Boxes All sites in an Armada are autonomous in their actions, but required to abide one simple rule. *Data may never get lost by dropping it on purpose.* This restriction forbids any site to remove data without actually checking if it would make that data unavailable. Hence, within this contract, data *can* be dropped, if and *only if* a clone of that data can be found, which can handle the same queries. This effectively keeps the data available in the Armada. When a site wants to drop a box, it has to find out if the data it drops can be found somewhere else in the Armada. The best way of doing so, is to examine the local lineage trail. Clone operations in the parent trail indicate a replication of a superset of the data contained in the local box. Following such clone operation is necessary to determine if the clone is still available in the runtime state of the Armada, as it may have been combined with another box again, or simply be dropped or unavailable as well.

Functions In contrast to chunking functions, cloning functions are defined to have “overlap”. More specifically, the clone function puts no restriction on tuples, instead accepts every tuple, regardless which box it is on. Of course this is only from the “fragmentation”-like point of view. Additional actions need to be taken to get the behaviour of duplicating the data. The clone function is the opposite of being pair-wise disjoint: on updates it always also redirects to its partners in the clone contract. The clone function implementation includes the knowledge that a full replica of the data is available elsewhere. Due to this definition the implementation space for clone functions is limited to a single implementation of cloning behaviour.

Unlike chunk and clone functions, the combine functions are designed to reduce number of active boxes in the Armada. Where chunking and cloning facilitate growth, combining facilitates reduction by merging two or more boxes into one. By definition, this is done in such a way that duplicate data is reduced to a single copy. The resulting data is the merger of the data that is combined. The combine function takes the union of two or more other functions, and filters out duplicates from the data. Since the implementation of this function is a simple boolean OR relation between the functions it combines, there is just one implementation, like with the clone function.

Opposite to the clone and combine operations, the functions to use with the chunk operation are numerous and diverse. All but one of those functions are horizontal fragmentation functions, simply because only horizontally the actual data is involved. Vertical fragmentation is based on schema rearrangement into separate sets of columns. In fully vertically fragmented database systems, such as MonetDB [7], further vertical fragmentation is impossible given that all relations are already fragmented to the column level. Hence, we focus primarily on horizontal fragmentation, when we deal with chunk functions in the rest of this chapter.

4.3 Chunk Functions

It is not uncommon for databases to have pieces of the data that are accessed more often than others. Such frequently accessed pieces of the data are referred to as the *hot* sets in the data. The ratio in which the hot sets are accessed more often than the rest can be very large, resulting in a typical *hot spot* in the dataset. This skewed access may simply stress a server and/or make it sluggish to do queries on data outside the hot spot. In a read-oriented setting cloning the hot spot would improve the performance. However, in a setting with many writes splitting up the hot spot over two or more servers has the benefit of not having to synchronise the two servers, while it still improves the general performance of the system. This goes under the assumption that both machines are at least equally fast and network costs are equal. The effectiveness is dependent on the chunk function chosen to effectuate the split.

When a server simply runs out of (disk) space, further storage of data is impossible and service degrades. Actions have to be taken to release such server by chunking a part of its data to another server. While for the problem at hand (capacity) no performance issues are of relevance, splitting can occur based on pure space-wise grounds.

4.3.1 Function Types

With capacity and performance as driving forces behind fragmentation in mind, we identified the following chunk functions.

range function The range type of functions span a consecutive region of values. It requires that the values can be ordered, such that ranges can be defined. A unique and dense sequence suits the best with this function, to have ultimate control over the volume of the ranges. Without this characteristic, ranges can have a hard to predict volume. The function has obviously no means to control volumes with duplicate values. However, to a certain extent the range function can deal with duplicates, for as long as the volume of a range can still be controlled to be small enough to fit. The idea of the range function can also be extended into n-dimensional worlds by using cubes or more dimensional “ranges”. A special form of the range function is the divider function, where the range starts or ends at the borders of the encompassing scope, thereby just dividing into two consecutive regions.

hash function Hash functions take key values and produce new values that ideally are equally spread over the space of values produced. Typically the space of produced values is smaller than the possible space of input key values. Due to this characteristic, hash functions are not *reversible*: they are so called “one-way” functions that are meant to project one space into another. The actual chunk boundaries are defined on the result of the used hash function in the projected space. As a result, no relations other than the hash function between the input data and their fragmentation exist. Advantage of hash functions is that they do not require a sequential order such as the range function. However, it is hard to tell what the scope of input values belongs to a chunk. A good hash function does not produce a skewed result, however, with duplicate values, the result contains at least the same number of duplicates.

omega function The omega-storage structure is defined to divide based on bit-level of the attributes involved [41]. Given the bit representation of the involved values, it tries to find the bit position that distinguishes the values the most when applying the split. However, because it operates on bit values, the domain of a key needs to be known upfront, since all values need to map to an equally sized bit string. Typically, such bit string is chosen to be as small as possible. In [41] the author does not elaborate on how non-numeric values, such as strings, should be used with the omega-storage structure. A possibility for strings would be to use some mapping from the string value to an integer value that can be turned into a bit value, such as a simple hash function. Because the division is based on the bit position, the split always produces two boxes. This could be extended into 2^n resulting boxes by using the n most distinguishing bit positions.

round-robin The round-robin fragmentation function is, unlike the previously mentioned functions, a function that does not operate based on the actual data. In the round-robin case, there is typically some external entity that row wise distributes the data over a number of servers, such as a load balancer. Because the actual condition on when to send what data to what server is external (and might even be completely random), there is no function that describes what data goes where. As a result, the best one can do with the round-robin function is to administer that there is more data in the siblings. This comes close the clone function, with the difference that the full data is not available on every box. Note that round-robin is only feasible for an append-only environment which is known to be unique by some key, for each insertion would otherwise have to be checked against the other descendants in order to guarantee the disjointness constraint. This constraint is just explicitly guaranteed to hold in the append-only case, but the Armada system itself can only get control over this by paying very high costs defeating the purpose of round-robin. Without this check, the function is a threat to the Armada strategy. Because no guided redirection can be made, the risk of a client dangling between two or more boxes exists.

projection function For completeness, we include the chunk function that does vertical fragmentation. Unlike the previous functions, the projection function does not operate on the values. This means fragmentation performed by this function is not based on the data in the schema at all. There is no relation to the data, and hence volume control through this function is difficult. Instead, the projection function fragments the schema, resulting in a vertical split of the columns in the schema over a number of boxes.

compound function The compound function is a pseudo function consisting of multiple functions. By definition of this function, it is only a mere convenience function that can be simulated by applying all the functions it contains in order. However, we like to include it here, to discuss the different possibilities that exist when combining functions. By combining functions, more complex spaces can be described, for instance multiple ranges. When the ranges are very small (a tuple) this might also be used for addressing "per tuple" spaces. Given this granularity possibility, a fragmentation based on a regular SELECT A, B, C FROM X WHERE Y SQL query could also be modeled in a compound function as a combination of the projection function with some of the other functions, given that the key is suitable for the selection. As a last note, the round-robin function can not be combined with any other, as it does no selection on the data, while still fragmenting horizontally.

Most fragmentation functions have some requirements in order to operate optimally. The range function can only be applied when there is some knowledge on the space it is being applied to. It makes no sense to divide a space in two if the separation is done in such a way that it does not divide the actual data, and hence leaves one box empty. It is necessary to have a clear idea on the "minimum" and "maximum" values in the space and what is between those.

The hash function is less selective than the range function, but depending on the hash function itself, its output can be skewed. Having the right function for the key values is a complicated task, and belongs to the standard hash research problems. In order to make a good hashing function, it is necessary to know what the input space is, and whether it is skewed already.

4.3.2 Classifications

The aforementioned implementations of chunk functions can be categorised according to several classification schemes, based on some properties of the functions.

An obvious first categorisation is based on the fragmentation being made by the functions. Here basically two categories are to be distinguished: horizontally and vertically. An extra category for a combination of the previous two is also included for convenience. The horizontal fragmentation functions are the round-robin, hash and range functions. They typically make a row-wise fragmentation of the data, where tuples are either within the selection or outside of it. In the category of vertical fragmentation functions, only the projection function fits. It is the only function that fragments by means of splitting the columns in the table. Of course a combination between the projection and a horizontal fragmentation function can easily be made, resulting in both horizontal and vertical fragmentation. This is covered in the combination category.

Another classification considers how the chunk functions operate. Three types of chunk functions can be distinguished: functional, predicate based and simple/compound functions. The functional class uses some algorithm to calculate a new value (apply a function) from the given key value, which is the base for the decision whether the function covers the key value, or not. The only function that exhibits this characteristic is the hash function. Predicate based functions use a certain predicate value to directly decide upon the key value whether the function covers the key value or not. Functions in this category are the range and projection functions. While the projection does not deal with the actual data, it still can be considered a predicate based function if the columns are considered the key value for that function. The last category of functions in this classification is for the simple or compound functions. Functions of this type are either not doing anything at all, or a combination of other functions from other categories. The compound function is such typical function.

A more abstract classification is based on two axes of the different functions. The one axis considers whether the function is based on the data in the Armada

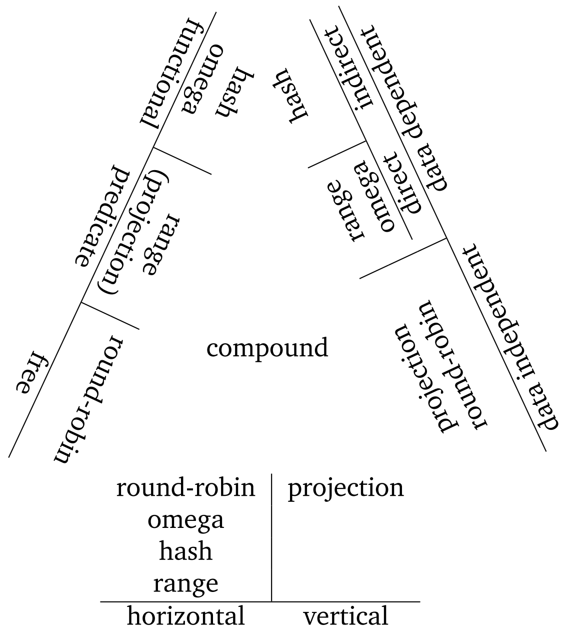


Table 4.1: Schematic representation of Armada function classification.

relation or not. Not depending on the data results in decisions that cannot be reproduced from within the Armada, as some external decision resulted in the function. The second axis is made for data dependent functions only. Data dependent functions can either depend directly or indirectly on the data. Direct functions apply some condition on the data directly ($f(v)$), while indirect functions first apply a (number of) functions before applying some condition ($f(h(v))$). This classification is depicted in Table 4.1. The categories are in the top half of the table, examples in the bottom part.

4.4 Function Trails

A number of functions applied in a sequence is called a *chain*. In Armada a trail is a typical chain of functions applied to the data. To determine the possible contents of a box, it may be necessary to examine the full trail. We refer to the possible contents as the “coverage” of a box, or function, i.e. that what it selects. In practice, the tuples that match the function.

A trail consists of steps, which each have a function associated to them. This function need not to be a chunking function, also cloning and combining functions are administered in the steps. Since chaining is about determining the coverage, the clone function can be left out of consideration in such trail, since it doesn’t change the coverage in any way. The coverage before and after the clone operation is the same.

An idea behind Armada is growth over time. This means that trails also grow over time. New steps get added to previous trails, and so a history of steps is being retained. As a result, also the functions in the trails are used on top of each other. In the chain of functions, the functions’ coverage can be based on the previous one(s), or fully self containing. The first one we refer to as *relative* functions, the latter one as *absolute* functions.

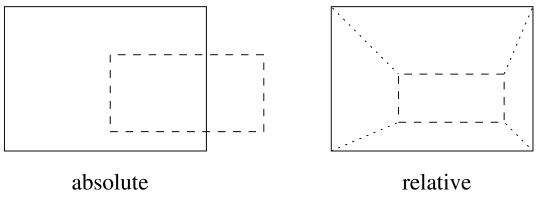


Figure 4.1: A graphical representation of absolute and relative function coverage.

Relative Domain Functions When dealing with relative functions, the coverage of the function is based on the coverage of the previous function in the chain. One could compare relative functions to percentages. Every new function works on top of the old function coverage as if it were 100%. By this definition, each new relative function is most of the time more “narrow” than the previous in the chain, but at maximum equal to it. This avoids redundancy like might be the case with absolute functions to get the right subset, for instance as a combination of different functions. The functions can be simple as they do not have to take the coverage of the previously applied functions into account. The right-hand side of Figure 4.1 depicts this relative coverage, where the dashed box is based on the outer box it is contained in.

Absolute Domain Functions The class of absolute functions have a fixed coverage based on the entire value domain, as observed by the origin. This makes the resulting coverage independent on previous functions, although it might change the actual coverage depending on them due to occlusion. In the most extreme case the function before the absolute function in the chain selects a part of the entire space that is not in the coverage of the absolute function. An example of a partial overlap of this kind is depicted on the left of Figure 4.1. Even though the dashed function coverage is within the outer box, it also falls partially outside of it. The result may be undesirable since the dashed function may expect values in its entire coverage, while obviously only the overlapping part receives values. This does, however, indicate the independence of the function with respect to its environment. In practice, it is most useful for absolute functions to be contained into the predecessor function, and so be more "narrow", like relative functions.

The advantage of an absolute function is that when functions in front of it in the chain are removed, its coverage stays the same, even though the received values need not to stay the same. Also when it is moved to another location, regardless whether the whole chain (trail) is copied or not, the coverage of the function remains the same. However, in the Armada model, this is unlikely to happen.

Function Composition A chain of functions, either relative or absolute, is a concatenation of multiple functions which need not to be of the same type. However, chaining arbitrary functions might not be possible or a sensible thing to do based on previous functions in the chain.

Absolute functions can be combined with relative functions, and vice-versa. Because absolute functions in principle do not take into account the coverage of the predecessor function, they run the risk of being ineffective due to non-coverage, as mentioned previously. Two range functions can easily be positioned in such a way that there is even no overlap between the two. However, range functions are an easy example, since they allow to easily see their overlap. Hash functions, on the other hand, are much more difficult. Not only is it harder to visualise their coverage, but also is the way in which any other function can be applied on top of them more difficult. Due to the characteristics of the hash function, it is not known what the relation between the tuples selected by the hash function is. Hence, it is hard to determine what the coverage of the function is, which makes it in turn hard to apply for instance a range function to tackle a skew problem.

4.5 Self-Management

The ultimate self-maintaining Armada system applies the clone, chunk and combine operations automatically upon need. The problem here is not *how* to apply, but *when*. The Armada model specifies *how* to effectuate an operation, but not at what conditions (*when*) using which function.

Cloning Cloning is used to add redundancy to the system. While redundancy in general is a safeguard for the system not to loose data, it can also serve as performance win due to load spreading. However, cloning adds the need to keep all clones synchronised. Therefore, cloning in itself is an operation that remains expensive after the operation has been applied. In some cases, a performance problem can be solved using chunking as well. Consider a hot-spot created by point queries, splitting it up in two distributes the load as well. The same hot-spot, but created by range queries would not reduce the load in terms of query hits, but perhaps it does in terms of the amount of data each of the boxes have to use for answering. Eventually, the entire hot-spot may also be chunked to a new and faster machine that can handle the load more easily.

While chunking causes much less overhead after the operation has been applied to maintain the chunk operation, it is preferable for an Armada system to use chunking over cloning, where the consistency of both clones needs to be maintained in some way or another for it to be effective. With this insight, the decision to use cloning unavoidably has to come from outside the system, e.g. as a (human) preference. Only active boxes are eligible for such operation, hence the further the Armada is chunked into pieces, the more clone operations need to be applied to clone the entire Armada system.

Properties A more complex solution would be more geared towards ease of use relying on the flexibility of the Armada system. Here, the tree based structure is used to define an inheritance based property system. The properties define boundaries for the Armada system to operate within. The more properties defined, the more restricted the systems in the Armada are regarding their autonomy. Two properties for cloning would suffice to balance the requirements from outside the system with the autonomy inside the system. These properties are the minimum and maximum levels of replication that the Armada system should maintain. The maximum level implies that the system has a freedom to clone more than the minimum level dictates. This implication means that the Armada system is able to find situations where cloning has benefits over chunking, even considering the burden of the continued maintenance to keep both clones in sync.

In this complex solution, at any time, on any box in the Armada, the set of properties can be changed. Such change triggers addition or removal of clones, to be automatically performed by the Armada system. However, the complex structure that the lineage tree can become makes this rather difficult to achieve. First, in a solution like this one, the inactive boxes, such as for instance the origin, are also valid boxes to control the properties of. This results in for instance creating a clone of the origin box, representing the entire Armada. Second, due to the inheritance of the tree, it becomes less obvious what actions are performed if at various levels in the tree properties are changed. For instance, a box may have inherited the maximum replication level, but its minimum replication level explicitly set. There are more solutions for what should happen if the inherited maximum replication level is changed to a value lower than the minimum replication level. Lastly, it is difficult for an Armada to know that the properties set on the boxes are satisfied. On the one hand, the sites hosting the boxes aim to remain autonomous and not to get involved in contracts with others. On the other hand, a box that inherits a minimum replication level needs to know whether it should clone itself, or whether the predecessor has already taken care of this. Eventually only the active boxes can perform the clone operation as requested, which means that in an existing tree it can only be determined how many clones of a sub tree exist, by traversing it down and building an image of which parts are cloned and how often. Combine operations in this scheme make it even more complicated since arbitrary boxes may be combined, and hence it may be impossible to deduce what part of the tree is still redundant and what part not due to the nature of some chunk functions.

Chunking A self-managing objective that an Armada system can take, is to fragment automatically when capacity problems occur. This is a natural objective, as without fragmentation, the system is limited to the size of a single system. Hence, the target of the objective is to allow the system to grow beyond the limitations of a single system.

Using pre-allocated small chunks, fragmentation does not apply any more like before. Since fragmentation is applied in advance to create the small chunks, a new chunk is allocated on another system when it does not fit on the original one. Here the slack space profile setting is aligned with the size of the chunks. A chunk is not assigned if there is not enough free space left. Key problem with automatic chunking is what function to take to achieve this effect.

A solution that avoids creating special functions for the problem at hand, is to chunk the data upfront in small chunks. Many of these chunks fit on the same system. However, once the system overflows it is obvious what chunk should be placed somewhere else. With these small chunks it is also easy to define the "load" for each chunk, to possibly detect a hotspot this way. With this knowledge, such chunk could be cloned, or chunked such that the box is essentially moved to another system. The drawback of using many small chunks is that the Armada itself grows artificially fast because of all the mini chunks allocated on the same system. This stresses the lineage trails, and their effectiveness on large scale.

To create less lineage "overhead", a chunk operation is only performed once capacity problems demand so. Chunking in this case means that a follow up has to be found for the current box, such that additions of data can continue. Without any knowledge of the data inserted in the box, the best way to chunk that box is by splitting it in two equal pieces. Equal here can be seen as in "data coverage" or as in the number of tuples the two resulting boxes contain. While the latter achieves a better distribution of the data over the two new boxes, the former is easier to implement using a range function, typically by taking halfway between the minimum and maximum values. By design, some hash functions are well suited for making an equal distribution of the original box. However, since the contents of the box differs per case, it is hard to tell if the chosen hash function also distributes the box contents evenly.

Statistics Assume a range-based scenario, where range functions can be applied. Ideally, when a box is chunked purely because of performance problems, the hot-spot in the box is detected. This can be based on statistics kept by the database kernel of frequently accessed ranges derived through e.g. a simple histogram. Depending on the histogram, one or multiple hot-spots may be found, or no hot-spot at all. Instead a close to even distribution may be found. In case of a single hot-spot, the chunk function can be designed in such a way that it splits the hot-spot, or that it extracts the entire hot-spot to relocate it to another machine. The latter option may be feasible if the Armada system has reasonable beliefs that the other machine is able to handle requests to that data more powerfully. This can be e.g. due to other hosted boxes, or physical hardware configuration matters. When multiple hot-spots are found, each can be assigned to a separate machine, or perhaps multiple combined into one and moved together.

In case of an even spreading of load, e.g. when the histogram shows no data that is obviously accessed more often, the entire box can be simply seen as hot-spot and hence split in two even pieces. However, in such case also a hash function could be beneficial due to a more even distribution. Though, since the result of a hash function is not a consecutive sub-range of the original space, it is possibly less effective as a range function if there are many range queries performed on the data. When a range query fits in one of the sub-ranges of a range function, it can be executed on a single box, whereas with a hash function this is not possible. However, once the range selection spans both boxes, there is no clear benefit, since both boxes need to be consulted, as is the case for a hash-function.

Storage Capacity When a box runs out of free space, said box needs to be chunked to resolve a capacity shortage problem. Detecting such shortage usually is fairly simple given that there is a notion of available "free" space. After this has been detected, a new box has to be added to extend the system. One option for that is to chunk the existing box in such a way that all data remains where it is, while new data ends up in the newly allocated box. This only works in certain data scenarios. First, the current contents of the box has to be described. While minimum and maximum values can be used for that, this may not be sufficient to achieve a split where new data goes into the new box. Randomly inserted data, for instance, is hard to describe. This highlights the second required point for this method, which is the need for consequently added data, as typically found in log-like data that is append only and fairly suitable for range partitioning.

Many other scenarios are not suitable for range partitioning this way. Instead they may have gaps in their keys which can be filled in later. For these scenarios it is better to free up some space on the original box, such that insertion of data on the original box is still possible. The free space allocated, the slack space, depends on the chunk function chosen. The amount of slack space available, influences whether the original box has to be chunked again later on when it reaches its capacity limits again. The easiest way is to simply split the entire box in the middle, to effectuate a slack space of 50%. This amount is the best one can do in a scenario where it is unknown how and when the data is inserted. If there is some information available on how the data is inserted, the slack space level can be adjusted to suit, such as shown before with log-like data insertion. Advanced statistics may record the amount of "appends" versus "inserts" given a range ordering, as to define an efficient slack space percentage. If slack space is allocated and never used, the result is unused space in the Armada system.

Summary

An important aspect of the Armada model, is the function as part of an operation that defines how data is spread. The amount in which this function can be devised by a site, defines how well such site can operate on its own, and hence how well incremental scalability within the cluster is effectuated.

Functions are not always independent, the chain of functions of which they are part in a lineage trail, may put certain restrictions on them. A limited amount of function types can be identified to be used, with their own properties, affecting the usefulness of other functions to be applied in a chain.

Thus far, the decision on a certain function is best done based on simple heuristics, until further research shows better ways. This way it is easiest to avoid clones in a system, and to apply chunk functions of a range type, based on histogram information of data usage frequencies.

5 Execution Model

5.1 Introduction

The Armada objectives explicitly put initiatives outside of the world made of boxes and sites. On purpose, the user of the Armada system is expected to play an active role in the process of its own query execution and the overall state of the Armada itself. The emphasis is put on the capability of resolving problems by making decisions. This is typically the user, but assisted by a program that performs the execution of queries, hopping between servers to collect pieces of the answer and construct the final answer out of those. Additionally, when clones are in the system, synchronisations between them are the responsibility of those who add or modify data, for which an application can perform a by the user chosen strategy.

When the user in charge of all decisions, the system puts a heavy burden on him or her in terms of interactions, and hence this is not desirable. As seen in Chapter 4, operations can be automated to a certain degree, thereby relieving the user from work. Also here, many actions the user should take can be supported by a program that acts on behalf of the user.

This Chapter presents some techniques that can be applied on top of the Armada model. It shows by simple solutions that the setting where servers deny certain responsibilities, is not unusable.

Assisting Users Assume a user tries to insert 10000 tuples into an Armada. Sites in the Armada can hold at maximum 1000 tuples. Without any help, the user would start inserting tuples on site S_0, which in this example initially is empty. After 1000 tuples, S_0 reports to be full, and something has to be done to resolve this. The only option the user now has is to to start looking for a new site to host more of its data. If the user fails to do so, it is obvious that it cannot continue inserting its data. Given that S_1 is free and the user found it, it instructs S_0 that it should chunk to site S_1 using a given function. S_0 may refuse this operation, for instance if it cannot reach S_1, or when S_1 is not willing to cooperate.

Obviously this continues with the user involved all the time for the rest of the chunk operation and further inserts, resulting in an Armada eventually causing more headaches than delights. Instead, an *agent* in the system can help to alleviate most of the work for the user. Redoing the same example with the agent as mediator between the user and Armada sites, the agent receives that S_0 is full and starts searching for a new site on behalf of the user. Agents can use economical models, statistics, first come first serve strategies, etc. to select a site to chunk to from a pool of existing free sites. If the agent fails to do so, it can return to the user with the problem at hand. The agent lets site S_0 decide how to chunk to S_1, which includes the moving of data from S_0 to S_1, if any.

The agent then continues to insert tuples on S_0. Depending on the chunk function chosen, S_0 can handle those tuples, or not. In any way, it has the means to redirect the agent to S_1 when the data does not fit any more. Once the agent receives such redirect, it continues inserting data on the new site. Eventually, the cycle repeats itself when S_1 reports itself to be full. In an ordinary case, the user need not to be involved in the entire process of chunking the data, when an agent is in effect. As obvious, most of this work that is the implicit responsibility of the user can be performed by the agent based on some heuristics, and preferences of the user.

Query Execution After the user's data has been inserted, the Armada consists of a number of sites. A user can contact any of those sites and request it to execute a query. In the trivial case, a user poses a query at site S_x which is local to the site itself. This is the case if for example the user requests a single key-value. In such case, S_x can simply return the answer to the query, and it resembles regular database query answering. The simple opposite of the previous scenario is when the user has a query that does not address any of the responsible boxes of S_x. In this case S_x returns one or more redirects to the user as an indication where to go. The user uses the redirects to continue its query.

A complexer scenario is when a site is only able to handle a query *partially*. In such case the coverage of one or more of its boxes address a part of the query, but not completely. Suppose the user sends S_x a query which it can only partially handle. S_x returns the results for the part it can handle, accompanied by the sites that the user should try in order to complete the result. The user can traverse all redirection sites to try and complete the full result for its query. It has to keep track itself which sites have been seen, as the sites it contacts do not know which sites the user already visited. As such they also return sites previously visited by the user together with the partial results they provide. This scenario depends very much on the user to keep track of the query process, without having access to the actual parts the query consists of. In particular, clones in this scheme may confuse a user that only knows what sites to visit.

An alternative to the previously described approach is again to use the agent for the execution of a query. Instead of having the user being confronted with a "query plan" returned by a site its given query, the agent keeps. The query plan consists of sub queries that have to be executed to finish the query. Because the sub queries match boxes, the agent simply starts traversing the query executing the single box queries, and glueing the results together. In some cases this may result in a large result, or a very lengthy execution time. Since the agent keeps the administration of the query execution process, it can present its progress to the user, who can additionally also control further execution in such case.

5.2 Query Resolution

An Armada system contains sites that host boxes. Sites are database powered entities that perform the local tasks for each box they host. They handle requests either by sending data or by sending a redirect. As noted before, a mediator between user and site is available in an Armada system, called an agent.

Humans are still indispensable for the existence of the Armada system. While many types of users exist, for the system the humans are those that can physically influence the system. That is, adding new hardware to the system by means of new sites, or extended capacity. Even though the system could order new components itself, it physically has to have them installed. But also in the use of the system are humans those that can resolve conflicts, by just making a decision which the system itself cannot devise. Input on what parts of the Armada system to clone reflect a human preference for redundancy. Last but not least are the data and its queries that originate from human users, but essentially make up the existence of the Armada system.

Agent Tasks The Armada agent can independently do some work for the human user. Typically, an agent carries out query execution and follows redirects. As long as there are no problems carrying out its work it is not necessary to bother its user with the work of following the redirects and continuing the query resolution process. To help even more, the agent tries to construct a full answer when feasible, thereby hiding the distributed fashion of the Armada system, and performing the database operations to construct the final answer out of the parts that it has received. Finally, when the user adds data to the system, the agent can take care of performing chunking transparently during the insertion process as long as sites with free space are available.

The agent needs the user in cases where it cannot possibly act itself correctly. It is up to the user to decide what to do with a query for which one or more pieces of that data are currently unavailable in the system. For long running queries, the intermediate results may be of interest to the user. Incremental query results based on chunk fractions allow a user to abort execution after it has only partially been processed, while still having a partial answer. Obviously this is entirely a user-based decision. If there is no free space in the system, the user needs to come to an action to either stop adding data to it, or to make space available.

5.3 Consistency

In principle the Armada system does not advocate a master/slave setup towards clones. By not doing so, the risk of the entire system to rely on a single master is avoided, as well as that a single update bottleneck is absent. The consequence is a multi-master setting, where each clone can accept updates, independently of others. Synchronizing the independent updates can cause conflicts when the same data is updated at two or more clones, when a uniqueness constraint breaks because of two independent updates inserting a same key, or when data is deleted which happened to be deleted on the other clones already.

Keeping the clones synchronised with each other is a tedious job. The process of synchronisation can be troubled by a number of situations in the system as indicated before. First it has to be determined if, and if so, how many clones there exist and where they are located. In principle this can be solved by looking for clone operations in the predecessor trail. However, there may be multiple clone operations, and those encountered may actually be based on outdated lineage trails, and hence cloned again, or even combined thus effectively removed.

Synchronisation Agents need to deal with previously mentioned conflicts and possible non-directly solvable delays. A danger of pushing the clone synchronisation task to the agent, is in the matter in which the agent takes its job seriously. The system risks synchronisations not being done, just because the agent neglects to do the job. However, if the Armada system would perform the synchronisations, it runs the risk of having to make decisions on conflict situations, as well as ending up in dealing with unavailable sites.

To address the risks of the Armada system, it has to be guaranteed that synchronisations eventually can be made. This means agents need not to do their job, while there is enough information in the system to cover up for that, such that other agents can do the synchronisation, if necessary. For this to work, each clone in the system needs to keep track of its own transactions. That is, it needs to store a counter that identifies the current state of the data. Next to this counter, it also needs to be able to produce a delta between two counter values. This may be implemented e.g. via the transaction log.

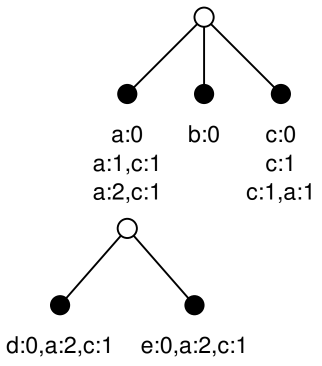


Figure 5.1: Multi master synchronisation counters.

Counters Upon creation of each clone, a new counter is initialised. However, to keep track of the state of all other clones in the system, this counter is not a simple single value. Consider Figure 5.1 where two clone operations are depicted. The first clone operation resulted in three clones, the second clone operation cloned one of the clones into two. Consider the first clone operation. Each clone here gets its own counter, which is uniquely identified. For readability purposes, we chose the simple values *a*, *b* and *c*. A unique identifier can easily be deduced from the path back to the origin for each clone. The initial state of the counter is zero, which is appended to the identifier, separated by a colon in the figure. If both a and c perform a transaction, their counters are incremented indicating a changed *state*. The result is an increased revision number in the counter, thus: $a : 1$ and $c : 1$. b remains in its initial state, as it has not been changed.

Now consider the merging of the transactions among the three clones a, b and c. When a is merged to c, c needs to process the single transaction that was applied to a. In the most trivial case, this transaction is not conflicting at all. Applying the transaction hence is simple and c would be up-to-date with a. However, even though a transaction does not conflict upon merge, it can yield in problems after the merge. Assume transaction $a : 1$ consists of $p = 5$ and that transaction $c : 1$ consists of $p = 3$. Now regardless of at which point in time both transactions were performed, because a is merged to c, the value of transaction $c : 1$ is overwritten, resulting in $p = 3$. While the decision whether this is correct or not is up to the user, we are here concerned over the consistency over the clones once c is merged to a. Because of the transaction $c : 1$, a needs to be updated with this transaction. Merging it as before would yield in $p = 5$ on a, since $c : 1$ contains this. Obviously both clones, while they are fully merged and synced, are not equal. Needless to say this is an unwanted situation that needs to be resolved.

Revisiting the merge of a to c, once c has applied transaction $a : 1$, it needs to consider this as a sub transaction of its own *current* transaction as indicated by its revision counter. This makes sure that the *blind write* $p = 5$ is applied in $c : 1$, causing a consistent image in further merges. Still, the counter is not updated. While this is unintuitive, it is fact necessary not to do so to avoid an endless loop of merge operations between in this case a and c. If c's counter would be incremented, the merge of $c : 2$ to a would be of no problem, but the result of that merge would be $a : 2$, which then again would have to be merged to c, which obviously has no extra information and causes an endless loop.

Merge Administration So far we ignored how clones can determine what transactions they have merged from the other clones. To do this, each clone needs to administer which transactions from other clones they received. This is administered in the counter. Using the previous example where transaction $a : 1$ is merged to c, the counter at c becomes $c : 1, a : 1$, which describes its own state, and the updates from a it got matching a's state at that time. With this notation, it is also easier to accept that c's revision counter is not incremented by a merge operation. The counter is extended or updated with the revision number of a at the time of the merge. Using this notation, two clones are in sync if their counters are equal. However, this does not necessarily mean they are up-to-date.

With the extended counter form, merges can very flexibly be propagated through the system. Consider the initial state for a, b and c again. After transaction $a:1$ completes, it is merged into c. The resulting state of c becomes $c:0, a:1$. Now c is merged to b, this essentially merges over $a:1$, but b tracks it in its counter as a merge from c. Hence b's counter becomes $b:0, c:0, a:1$. Transaction $a:2$ can similarly be merged either directly to b, since b has $a:1$ in its counter, or via c again, without disrupting any future merge process.

The second clone operation in the example of Figure 5.1 is performed after the system has been running for a while. The revision counter of a is $a:2, c:1$ at the time of the cloning operation. The latest revision of a is not yet merged over the entire system. The clone operation produces the clones d and e, which get their own counters. However, they are copies of the original a, hence they inherit a's transaction log. This allows them to bring other clones in the system up-to-date with the last transactions of a. Since a ceases to exist after it has been cloned, no further transactions are applied to it. Still, d and e are part of the original a. Therefore, both new clones retain the last revision of a in their own counter, such that other clones can sync up to the last changes of a before it was cloned. Hence, the new clones start with the counters $a:2, d:0$ and $a:2, e:0$. For the system the new clones are equal to the others, and a appears as a clone that is never updated any more. Since it does not exist any more, it also does not require to be updated.

Clocks Our described counters strongly resemble vector clocks [50]. These clocks are an improvement of Lamport's virtual time concept [44]. Here time is reduced to the simple notion of "happened before", which means as much as for each two *events* a causal relation exists (a happened before b) or they are unrelated, which says nothing about when the events happened in a physical clock world. The causal relations are caused by messages between processes, in Armada's case propagations of updates. In the vector clocks, each process keeps a vector of counters, for every process in the system a counter in the vector. Messages between processes transfer the entire clock vector, such that not only the counter of the sending process is transferred, but also those of that were once received by the sending process.

It is not hard to see that in our described system of transaction counters, we also have vectors keeping the counters of the other sites in the Armada. Essential difference here, however, is that our vector are not of a fixed size, and it is not known how large they have to be. In fact they may differ per site. In practice this does not change much in the vector clocks, since an absent process could just implicitly mean a 0 in the vector. The clocks do not require to know the total amount of processes to determine if some vector describes a time that “happened before” another vector.

Summary

A user of an Armada system is required to be an active participant in its own query execution. This is inevitable for an autonomous and distributed system like Armada, where the high level of autonomy is only retained if clients take their own responsibilities. For the most basic tasks, a user can rely on an agent in the Armada system to help. The agent performs trivial tasks like following redirects and constructing query results. As long as no problems occur, such as unavailable sites or conflicts, the agent can do its job without asking the user.

With Armada pushing actions involving other sites to the user of the system, in case of clones in the Armada, the users need to maintain the consistency of the clones. For several reasons users may end up not doing their duty, and the agents can fail as well given they cannot resolve problems. A system to guarantee convergence is applied to all clones in an Armada. It relies on users performing actual updates, but it keeps the administration on what updates local on the sites. This system that keeps transaction counters on each clone, allows to merge updates from any clone to any other at any time, eventually reaching a globally consistent state.

6 Performance Analysis

6.1 Introduction

The power of the Armada model to point to the appropriate direction from each point in the system, yields in a converging search for data. However, instead of reaching the data directly after a single catalog lookup, multiple steps can be necessary to reach the data being looked for.

With an Armada system growing, the lineage trails grow along in size. The effect of such longer trails is the growing probability of needing more steps to reach the data being looked for. In this chapter we study the process of following the redirects from the Armada model. We focus on the costs associated to the process of following for different Armada systems. Since a traditional non-distributed system would have direct access to the data in any case, in comparison an Armada system introduces extra work caused by the redirects. As this is a given, next to identifying its costs, we experiment with different approaches to minimise their effects. The content of this chapter has been presented as workshop paper [31].

Throughout this chapter we frequent the terms *agent*, *site* and *box*. While these terms are defined in previous chapters, their definitions may be blurred due to various usages. In this chapter when we refer to an *agent*, we refer to the entity in the system that interacts with the data nodes (sites) in the Armada system. A *site* in there is a data node, capable of storing boxes in the Armada system. A *box* is a logical block of data hosted on a site, being either active or inactive. An active box points to data on the host site, an inactive box points to one or more other boxes that should be searched instead of that box.

run	PostgreSQL		MySQL		MonetDB	
1	0.131	28.415	0.099	22.159	0.143	30.824
2	0.135	28.440	0.098	22.117	0.116	30.998
3	0.147	28.435	0.098	22.162	0.118	30.364
4	0.129	28.435	0.099	22.113	0.118	30.192
5	0.130	28.421	0.099	22.122	0.119	30.182
avg	0.134	28.421	0.099	22.135	0.123	30.512

Table 6.1: Wall-clock times in seconds for performing 1000 queries over a single connection (left) or over separate connections (right).

6.2 Connection Costs

Each connection that is made to a database has some overhead caused by handshakes as part of the initialisation rituals on both the protocol level, as well as on the TCP stack. Table 6.1 depicts the results from a small experiment conducted to show that creating a connection to a database is an expensive operation. In the Table, the wall-clock times for performing 1000 “SELECT 1” queries using a client tool in seconds are shown for three different Open Source database systems. For each database, the left column shows the time it took to perform the thousand trivial queries over a single connection, while the right column shows the time for the same queries, but each over its own connection using a new client tool invocation. While the numbers are bound to the used software versions, the table clearly shows that creating connections is substantially more expensive than reusing the same connection.

In the experiment we tried to eliminate the overhead of query parsing, processing and execution, by taking a very simple “SELECT 1” query. The used operating system is OpenSolaris snv_101a on a AMD Athlon64 3800+. Performing 1000 executions of a very simple application (echo), to try and determine the operating system costs of executing the client utility averages to 7.3 seconds. This time is not subtracted from the right columns in Table 6.1. We used the 64-bits versions of the database software, PostgreSQL 8.2.7, MySQL 5.1.21_beta and MonetDB Nov2008. Subtracting the operating system overhead from the 1000 connections measurements, the latter still are 158, 150 and 189 times slower for PostgreSQL, MySQL and MonetDB respectively. For this reason it seems beneficial to try and reduce the number of connections an agent has to make during the query process, since this takes a substantial amount of time.

6.3 Metrics

Each measurement needs a way to describe the observed facts. Those descriptions are expressed using a given *metric*. Within our experiments on the Armada model, we are primarily interested in the performance of agents that navigate through the system. This performance depends on several factors, which we try to address given the following metrics.

hopcount The number of steps taken by the agent for a query from the starting site to the site holding the active box with the value being looked for. An agent that directly contacts a site which contains an active box responsible for the data value it looks for, has $hopcount = 0$ for that particular query. The bigger $avg(hopcount)$ becomes, the worse the seek performance of the Armada.

sitehits The number of hops to a site by the agent. When an agent makes a hop to a site it also performs an action on it. *sitehits* expresses the importance of a site by means of how many times it is hit. A query with $hopcount = 0$, increases *sitehits* by one.

sitequeries The number of actual queries performed by a site for the agent. This value is an indication for the query workload. When data or queries are skewed, the distribution of sitequeries among the sites shows a clear peak. A query is an action performed on a site, i.e. an insert, update or select.

sitetraffic The number of hops made by an agent from a given site to another site in the Armada system. Per site this can result in multiple incoming as well as outgoing hops, referred to as traffic. Frequently used traffic paths are an indication for classic bottlenecks.

sitefree The percentage of available space to store tuples. This can give insight on how well equally loaded with data sites are, and hence if the data is spread equally.

Further, we have the site capacity and box capacity. Also, multiple data boxes can be placed on a site, which allows to reuse a site (sitefree) if no empty sites are left.

It is seducing to think of the sites in the association tree as an in logical rings around the origin ordered system. As such, each ring is another step away from the origin. It cannot be said that the hop count to reach one site from the other equals the ring distance as the actual traversed path may take more hops given the tree structure of the association tree. More importantly, the ring distance only works as long as sites are used only once for storing a box. As soon as a site is "reused" the association tree becomes cyclic. With cycles in the graph it is no longer trivial to put sites on a ring. Apart from if or if not it would be possible to define a rule to still do the ring assignment, it becomes unclear how the resulting rings should be interpreted. Therefore any metrics that rely on ring distances are bound to become void once cycles appear. It is to be expected that cycles become a natural part of Armada to achieve a higher storage load, spreading throughout the available nodes and a way to gain access to trail information from other parts of the tree. The latter can help to reduce the hopcount.

6.4 Policies

To study the effects of various facets of a simulation, those facets need to be changed following a strategy to prove or falsify a given hypothesis. We identified four facets that are important for the performance of an Armada implementation in speed and space utilisation. For each facet, we identify a number of policies that define a certain action or behaviour for the given facet.

6.4.1 Chunk Policies

Chunking splits data from one box into two new ones. In our experiments, we use adaptive chunking. The chunk function is the divider function, that splits the original box in two. Further, for each chunk operation, one of the new boxes is placed on the site of the original box. As a result only one new site has to be found, and the amount of data that needs to be physically moved is limited.

With adaptive chunking, the chunking operations that are executed fully automatic are influenced by the environment as observed by the box. A simple idea is to have each box monitor the inserts being done. A simple "statistic" in this way can be to keep a boolean indicating whether the inserts done on the box have been "appends" only, considering a given (sort) order. If so, the divider can be set to retain very little slack space, which means for instance in a linear insertion case that the site load is high, comparing to the normal half min/max 50% slack space chunk function. If a box on a site chunked with no slack space has a new value inserted, this cannot be an append in sorted order any more (due to the chunk with no slack space) and hence, the append only flag is not set, so a regular 50% slack space chunk can be performed.

6.4.2 Trail Policies

During simulation, it quickly becomes clear that there are extensions possible to the Armada model that allow for more precise redirections. Often these extensions can be a part of the standard amount of trails being exchanged during a chunk operation.

Vanilla Armada Follow the Armada model unconditionally. Each site contains boxes, which have a predecessor trail, the self step, and the successor steps. This bare-bone approach forces each agent to go around in the Armada using the absolute minimum of available data.

Sibling Steps Upon each chunk operation, add sibling information to the sites involved. This is a small optimisation on top of the vanilla Armada model, as this adds extra trails which allows successors of a node to redirect to each other without having to contact their parent box. In practice, this means that the newly created site gets an extra pointer to its sibling, because its sibling already has this pointer via its parent, which is hosted on the same site. However, for the new site this also holds (there is a pointer to the parent box/site) and hence adding the sibling trail does not help much here: the new site can send the agent to the original site, which knows what the specific box is, which is irrelevant for the navigational structure. Therefore, this policy only makes sense when both new boxes are stored on a new site.

Agent Hinting Agents that are being redirected to an inactive box (hence requiring another redirect), hint the site that offered an out of date redirect with the updated (more specific) trail. The site can use that trail the next time an agents visits to possibly direct more accurately. The possibilities here are numerous. Agents can hint only for successor steps (only optimising the current sub-tree) or for predecessor steps (which quite often trigger another redirect) as well. Agents can only perform one level e.g. A to B, B to C, C to D, only adding D to A, or D to B, or both/all levels. Rationale here needs to be found with respect to the association trees.

6.4.3 Agent Policies

The Agent in an Armada simulation is the entity that performs most of the work. It basically deals with the entire traversing through the system, by means of following redirects. On successive queries, the agent has a number of options to try and minimise the amount of hops taken for each query. The baseline approach for an Agent is the naive strategy of the Lazy Policy.

Lazy A Lazy agent starts each query at the same site, which is the only site it knows, the origin. Obviously this stresses the origin with a magnificent hit-count, but allows the origin to exercise in redirecting to the right site at once (e.g. in combination with Agent Hinting Trail policy).

Random This type of agent picks a random site from the cluster for each query. It uses no knowledge whatsoever, but truly randomises to spread the load over the cluster. This avoids the origin being a hot spot, via a technique that could be employed by e.g. DNS load balancing. This approach stresses each site in the cluster for its ability to send the agent in the right direction.

Sticky Sticky agents stick to the last site they have touched, for their next query. This strategy is in particular useful when doing a linear insertion, as it most of the time yields in a $hopcount = 0$. Sticky is a very cheap policy that tries to do slightly better by not disregarding the last known state of the Armada.

Cache Smart agents cache the lineage trails they see when traversing the Armada, and use that cache prior contacting a site to make an educated guess what would be the most appropriate site to contact, e.g. the site closest to the target. Obviously, out of date cached trails can be thrown away when being encountered to reduce the search space. A caching client can ultimately get a hopcount close to 0, as its cached trails represent the part of the Armada it is interested in. However, this is an ego-centric policy of which no other agents benefit. It is not realistic to have all trails for the entire Armada cached, as this may be a large amount. This large amount may not be an issue memory wise, but it will be a performance issue given that the search space increases. Hence, the agent needs to define a policy for itself that defines which trails need to be kept, with a limited number of cache buckets.

6.4.4 Site Policies

A site in the Armada hosts boxes. The model does not limit a site to host just a single box. While hosting multiple (active) boxes is not difficult, the capacity of a site is no longer subject to the utilisation of a single box. In particular the process of finding a new site to host a box as result of a chunk operation is affected by the decision whether sites with available capacity can be reused or not.

Single Box Each site contains at most one box. If there are no more available sites, the Armada cannot grow any further, even though some sites may be hardly full storage wise.

Reuse Free Sites are used based on their available storage space. From the pool of available sites, the site site with the most available storage space is used, assuming all sites have an equal storage space.

6.5 Data Sets

The shape of an Armada tree is influenced by the data it contains and the order in which it was inserted. To experiment with different tree shapes, and to see the effect of them on the various metrics previously defined, we used the following carefully crafted workloads. For each workload we used the same value range starting at 0, ending at some predefined positive number. By doing this, the sets can be used to query the other sets without getting an artificial skew because of a range mismatch. This avoids either having a high skew on the edge node because all high values are mapped onto it, or having a skew on a range of nodes because the other nodes cover values not in the value range of the query set.

Linear The linear set is a simple ascending counter with regular gaps to fill up to the desired value range. Since no duplicates are allowed, each value appears at most once in the Armada. The gaps between the values are equal, and hence do not affect chunking decisions due to the introduction of skew.

Random A randomised list of values. Duplicates are forced to occur during the generation process. 10% of duplicates are generated and gaps are likely to occur. The duplicate values are randomly spread throughout the set.

Uniform Again a randomised list of values, but using a perfect even distribution. While gaps are still possible, duplicates are not allowed. The random order of the values, causes unlike the Linear set to have unordered insertion of values.

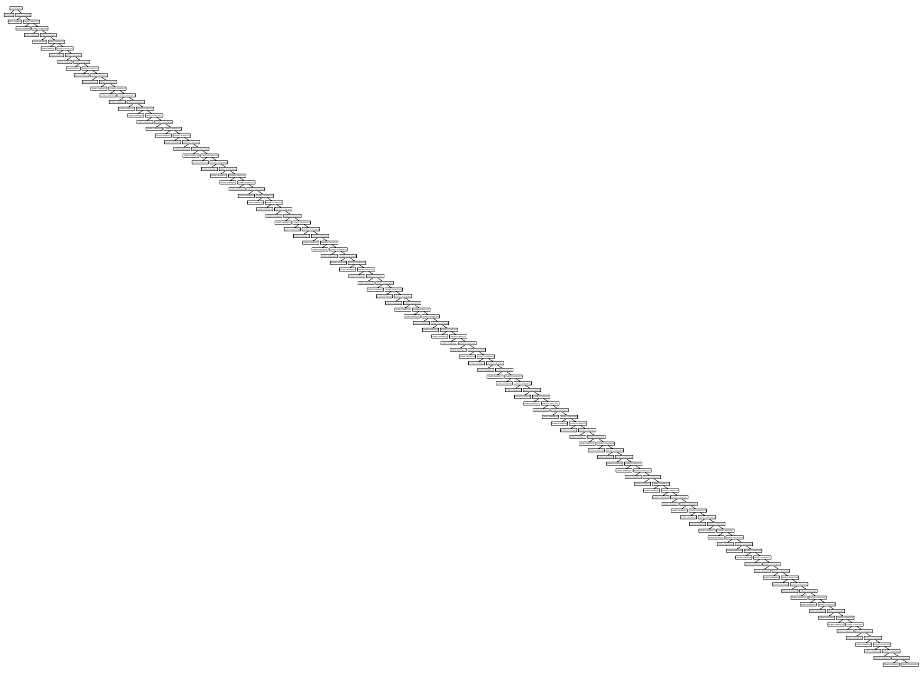
Zipf Another random input based set, but with a value probability following a Zipf distribution. In this set, typically a small amount of values are very popular, and likely to occur more than once, while other values hardly occur, if at all. This distribution clearly is very skewed. In the generated set we chose to have the popular items to be those with a low value.

Real World A set of integer values extracted from the entire MonetDB/SQL test set. Skew, duplicates and gaps are occurring here, as many tests use the same values, very close values and entirely different ranges. This set does not have the fixed range the other sets use, because the real world values simply are fixed due to being real. This set is hence only of limited use in comparisons.

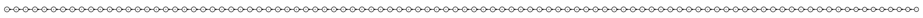
6.6 Data Loading

The previously described sets generate Armada trees with characteristic shapes when being inserted. Starting with the Linear set, the simple ascending nature causes the values to be inserted into the most recently created box, which is chunked once its capacity is exceeded. The lineage tree of this set eventually is a deep tree, where each time a chunk is performed, the last added box is chunked. Because per the chosen chunk policy one box is retained on the original site, and the other containing the overflow values on a new site, the resulting association tree is a simple chain of successive sites being attached to each other. See Figure 6.1.

Random and Uniform The Random and Uniform sets result in a balanced lineage tree, see Figures 6.2 and 6.3. As values are scattered, the tree is built by approximately filling all of its active boxes evenly for the value range they cover. The association tree on the other hand shows a pattern where the older the site, the more offspring it has. Since all sites represent for the time being at most one active box, each site can overflow similar to why the lineage tree grows in a balanced manner. This means any site can get a new site association. Hence, the longer a site is around, the more site associations it gets due to chunks.

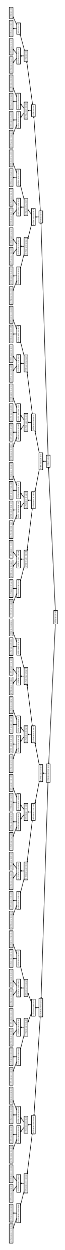


(a) Lineage tree

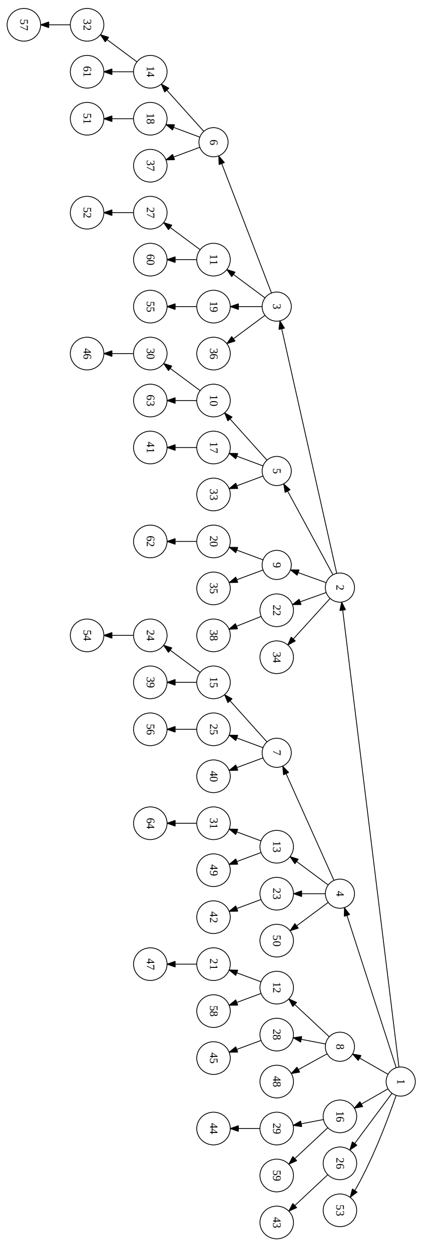


(b) Association tree

Figure 6.1: Loading the Linear set.

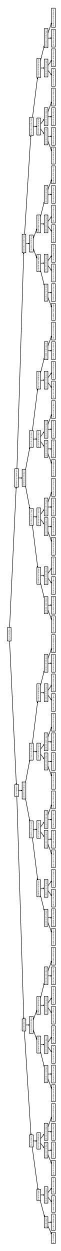


(a) Lineage tree

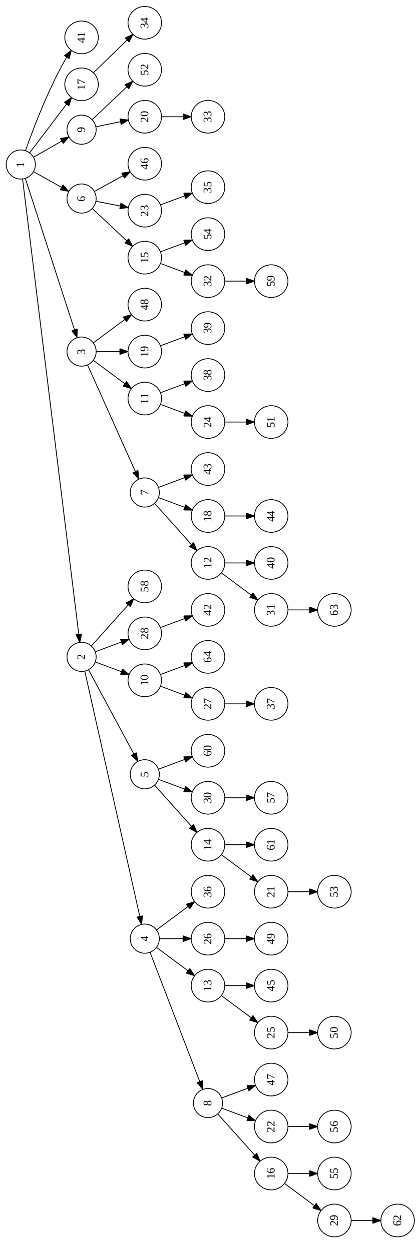


(b) Association tree

Figure 6.2: Loading the Random set.



(a) Lineage tree



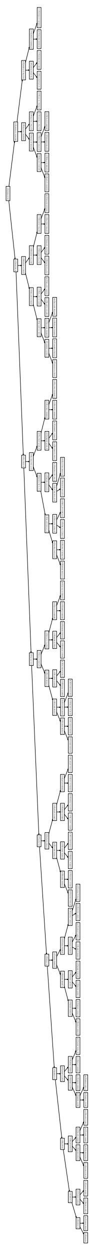
(b) Association tree

Figure 6.3: Loading the Uniform set.

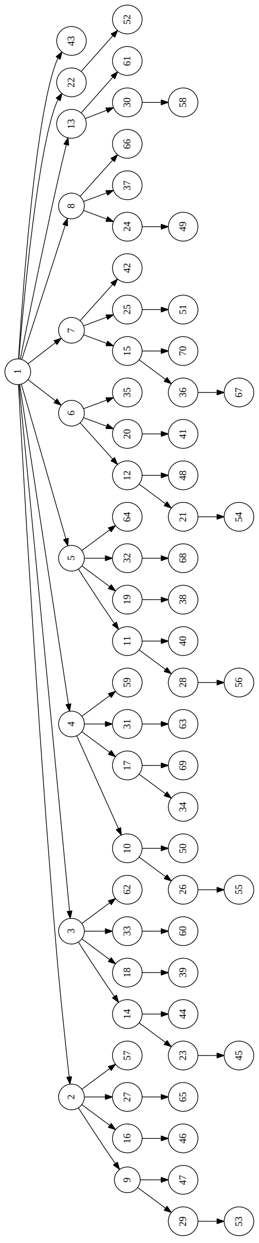
Obviously, the origin site has the most associations, with its direct offspring following in association count. However, not only its generation defines the number of associations, as also the age of a site is related. So typically what can be seen here is that each site has less associations to offspring itself, than its parent has, and within a generation every site has less offspring associations than the sites in his generation that are older. In total this gives a diagonal shape to the association tree.

Zipf In a Zipf distribution typically a few values are very popular and rare values are almost never used. The Zipf set we generated uses a Zipf probability for each value to occur. As random values are chosen, their chance of being put in the set is based on the Zipf chance of the value. This way the lower the value, the higher the chance it ends up in the set. The resulting set is skewed towards low values. This shows up in the lineage and association trees, see Figure 6.4. The chunk operator places the overflow values on a new site. Because small values occur more often, in the Zipf set, this means that more often a box needs to be chunked that is the non-overflow part of a previous chunk operation. This typically leads to a slight opposite of the Linear set, where the overflow part is constantly chunked. In the Zipf lineage tree, the boxes on the older sites are re-chunked over and over again, resulting in a deep, not very wide tree. Because there are also higher values, the tree is not as narrow as the lineage tree. For the association tree the re-chunking means that old sites have many connections. Because of this property the resulting association tree is very wide, and not so deep.

Real World The Real World set is based on data values harvested from the MonetDB/SQL test set and therefore not following any special pattern. The lineage and association trees show however that the set contains parts that follow the Linear set and parts that appear to have a Zipf or other high-skewed distribution, see Figure 6.5. This observation matches the nature of the set, which has many duplicate values as a result of slightly modified copied and pasted tests, as well as some linear sequence to just insert a sufficient amount of tuples with distinct values. In particularly interesting is the constant alternation between high and low values that occurs at almost every place in the set, except from a piece with a linear nature. The tail of this set shows an alternating pattern for low and high values that linearly increase, the high value much more than the lower value. In particular the hop count graphs that we discuss later on visualise this pattern very well.

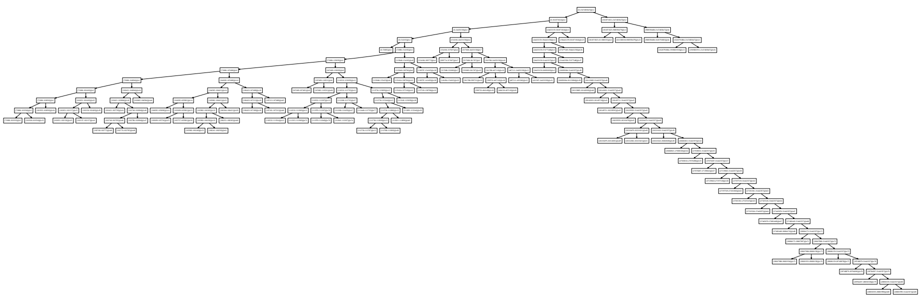


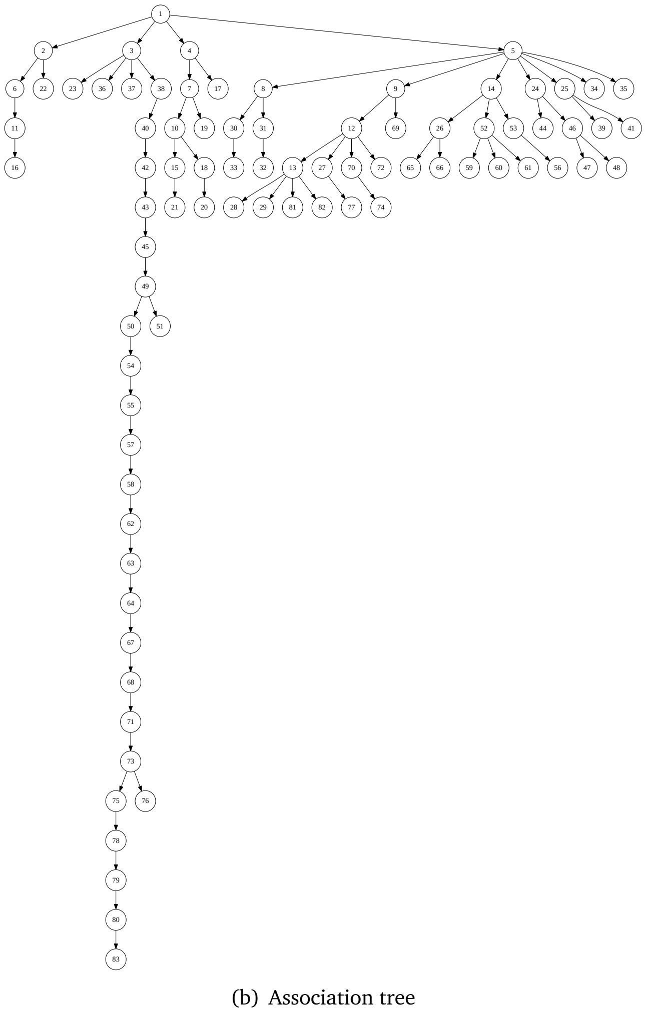
(a) Lineage tree



(b) Association tree

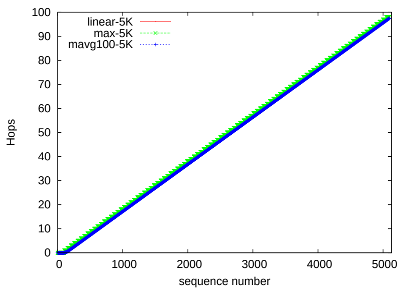
Figure 6.4: Loading the Zipf set.


(a) Lineage tree

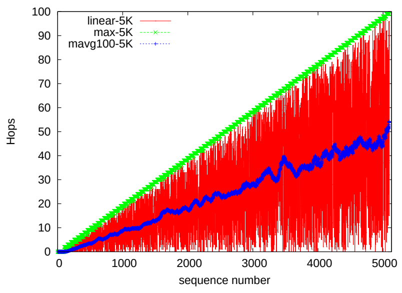


(b) Association tree

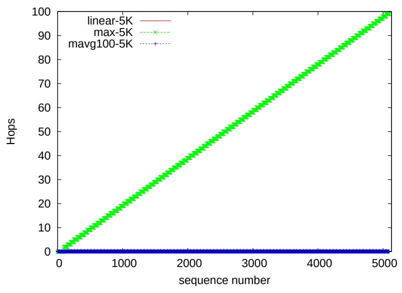
Figure 6.5: Loading the Real World set.



(a) Lazy policy

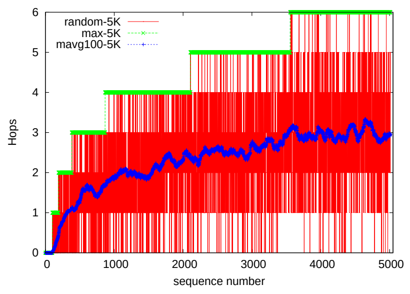


(b) Random policy

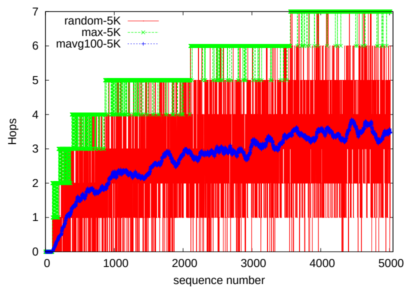


(c) Sticky policy

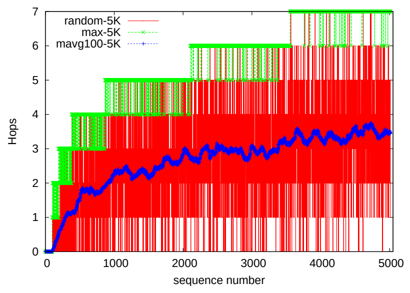
Figure 6.6: Loading the Linear set.



(a) Lazy policy



(b) Random policy



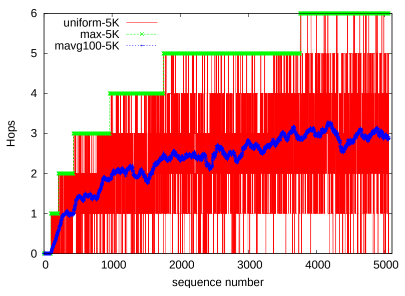
(c) Sticky policy

Figure 6.7: Loading the Random set.

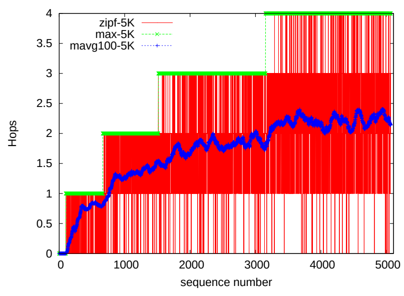
Agent Policies While the shapes of the lineage and association trees depend on the data set being loaded, the actual agent policy used does not affect the shape at all. Though, it influences the performance of the agent during the loading. There is not just one "best" policy, as efficiency in terms of a minimal amount of hops depends on the set in use. The Linear set typically has a "moving hot-spot" as it appends to the last added box. The Lazy policy here gets an increasing hop-count for every query after a chunk operation has been applied, see Figure 6.6. The policy that is well suited for this set is the Sticky policy, as it nicely "follows" the moving hot-spot, and needs at most one hop right after a chunk operation has taken place to jump to the new site. The Random policy performs better than the Lazy policy simply because it has a chance of accidentally picking a site closer to the moving hot-spot. As the policy is based on pure randomness the performance in terms of hop counts is on average half of the depth of the association tree. Lastly, the Caching policy performs on any set very well, simply because it has all trail information locally as soon as it has visited each location once. Because we assume an unlimited cache, this policy is always very well performing. Because of this we skip discussion of this policy for the other sets.

Random and Uniform The Random and Uniform sets cause an agent to be jumping back and forth between sites in the Armada, see Figures 6.7 and 6.8. Since this cannot be predicted, without a cache each guess is as wrong positioned as any other. On average, the Sticky policy does not perform any better than the Random policy for this reason. Interestingly, the Lazy policy is the best for these sets when inserting the data. The reason behind this is that because it always starts at the origin, it never has to jump back to the origin, as the other policies have to when they start in a wrong branch of the Armada tree. This makes the Lazy policy on average around half a hop shorter. This can be explained by occasional "luck" of the Random and Sticky policies when they happen to start close to the target.

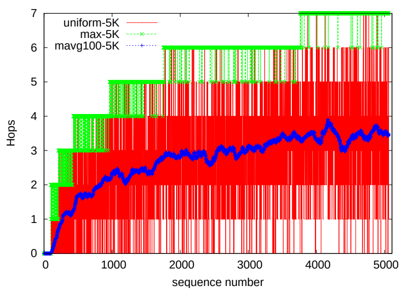
Zipf The Zipf set, even though it is skewed, shows the same pattern as the Random and Uniform sets, see Figure 6.9. This is not so surprising considering the Zipf set is also a random set, but with a Zipf probability. Because the tree is less deep, the advantage the Lazy policy has is bigger, resulting in almost one hop better performance than the Random and Sticky policies.



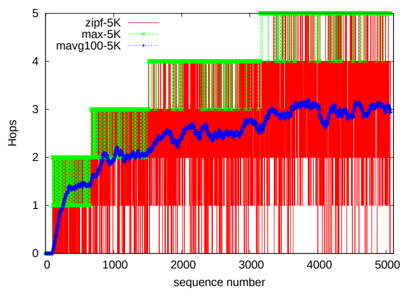
(a) Lazy policy



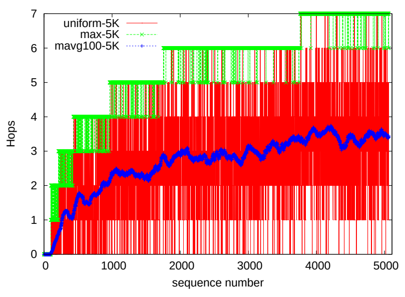
(a) Lazy policy



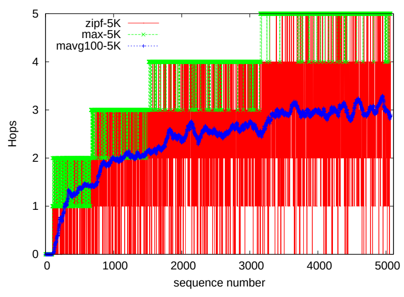
(b) Random policy



(b) Random policy



(c) Sticky policy



(c) Sticky policy

Figure 6.8: Loading the Uniform set.

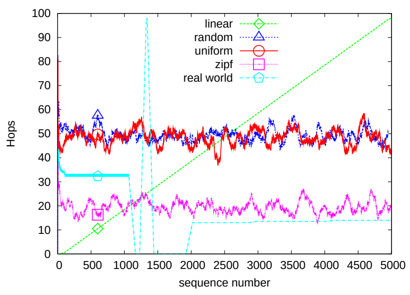
Figure 6.9: Loading the Zipf set.

6.7 Querying

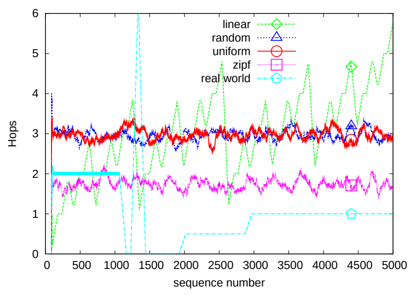
So far we have only observed the number of hops taken per insert during the loading phase. A constructed Armada tree can be queried again, as part of normal querying operations on an (existing) database. To further study the agent policies, for each loaded set, we query it using all sets per agent policy. Within the sets, not the same values have to be used. This means that when the loading and querying use different sets, values can happen not to be found. Though, a matching (responsible) site has to be found in any case.

Lazy Policy Querying the Linear set using a lazy policy in general yields in many hops. Obviously when querying with the Linear set itself the number of hops necessary per query continuously increases as the values are found further away, deeper in the tree. The Random and Linear sets have stable hop counts that on average are close to around half of the association tree depth. This nicely demonstrates that both sets are truly random given that the average is in the middle of the value spectrum. The Zipf set is as stable as the other random sets, but has much less hops. This can be explained by the nature of the set, where lower values are much more likely to occur than higher ones. As a result the average value is not in the middle of the value spectrum, but below it. Eventually the Real World set shows a pattern where first around the same values at on average 32 hops are retrieved for around 1000 queries, then some queries that do not require any hops, followed by a peak leading to the maximum depth of the association tree. After around 1000 more queries close to the origin, the remaining 3000 queries of the set are all around 15 hops.

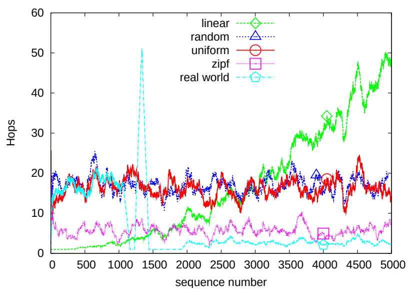
Random Policy When we switch from a lazy policy to a random agent policy, the number of hops taken per query are reduced by about 50% of the hops taken with the lazy policy. This can be easily explained, as the random policy on the linear set jumps into the tree, always in the right direction. In the same way that the random query sets on average target the middle of the value spectrum, the random agent policy starts querying on average in the middle of the association tree, which obviously cuts the maximum hop count in half. The Linear set using this policy shows random behaviour, but on average shows an increasing hop count to only half of the hop count necessary when using the lazy policy. Where the line in aforementioned policy was straight, in the Random policy it is slightly curved in a quadratic shape. This can be explained when considering the values being queried for. Since the Random policy on average positions the



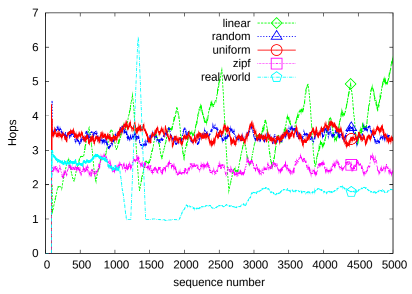
(a) Lazy policy



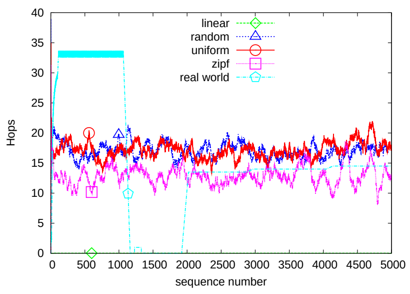
(a) Lazy policy



(b) Random policy

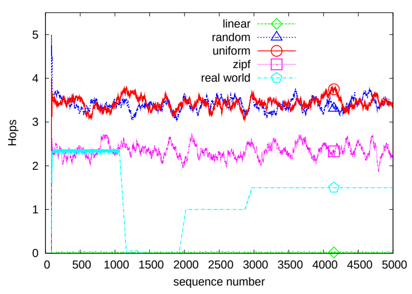


(b) Random policy



(c) Sticky policy

Figure 6.10: Querying the Linear set.



(c) Sticky policy

Figure 6.11: Querying the Random set.

agent in the middle of the association tree, lower values can be quickly found by jumping back immediately in the lineage trail as far as required, yielding in at most two hops. However, when values are requested that are deeper down in the tree than the start position, a one by one hopping down to the target has to be performed, with a higher hop count. The random effect makes this happen gradually.

Sticky Policy The sticky policy achieves the same hop counts for the Random and Uniform sets, because the positioning of the agent for those sets is like the random policy since the values queried are in a random order and hence end up at random sites. Since the Zipf set favours lower values, with the sticky policy, the agent more often ends up at a site close to the origin, resulting in a bigger chance than the random policy that the agent has to hop one by one away from the origin. The Real World set shows no peak using the sticky policy, as this peak consists of a linear sequence, which the sticky policy is able to efficiently follow. For the other parts of the set, the sticky policy results in higher hop counts, because the previous value is far way from the next.

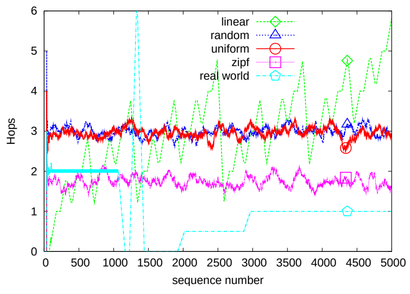
Cache Policy The cache policy obviously has very low hop counts, simply because it can position the agent very well for every query already starting once it has learned a part of the tree.

Random Set The Random set has much smaller hop counts compared to the Linear set, see Figure 6.11. This is due to the association tree depth of the Random set being much smaller as a result of better tree balancing caused by the random value insertions. Like before, the Random, Uniform and Zipf sets show a steady average hop count over all queries for the lazy, random and sticky agent policies. However, unlike with the Linear set, the lazy policy is the most efficient in terms of hop counts here. This difference of on average half a hop can be explained by the random positioning of the agent no longer being a jump in the right direction. The association trees for the Random, Uniform and Zipf sets are wide, and hence a random jump has a high chance of ending up in a wrong branch of the tree. The latter requires a jump back to the origin, which is the position of the lazy policy. Therefore the latter policy is on average more efficient on these sets. The Linear set, when queried on the Random set, shows different behaviour though under different policies. The sticky policy, as expected, allows to reduce the hop count during querying quite dramatically. The association tree built has due to the fragmentation function the property of sorted order, allowing the sticky policy to need at most one hop on a linear query sequence. Hence it again removes the peak from the Real World set.

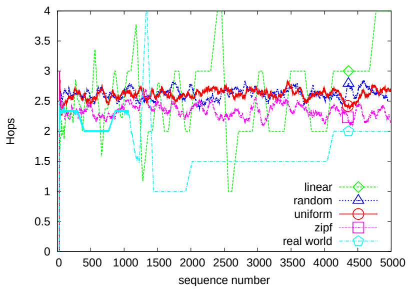
Uniform Set When querying the Uniform set with the other sets using the four agent policies, a slight variation on the loaded Random set is the result, see Figure 6.12. Since the Random set only has some duplicates, which the Uniform set does not have, this is not surprising. There are no noteworthy differences to be found, and the same remarks as for the Random set hold.

Zipf Set Because the Zipf set has an association tree with a smaller depth than the Random and Uniform sets, there are less hops possible in total. As expected, the Uniform and Random sets are again steady, but close to the Zipf query hop counts for all policies, see Figure 6.13. Because the depth of the tree is smaller, the positioning error of the agent is less punishing, hence in total less hops need to be taken. The higher values that are not in the Zipf set, are simply not found, but the responsible site is found earlier because of the smaller depth. Because the Zipf set is a random set, the lazy policy performs slightly better as with the Random and Uniform load sets. The Real World set shows a different pattern, since the lower values are more fine grained fragmented, and successive queries need different sites now.

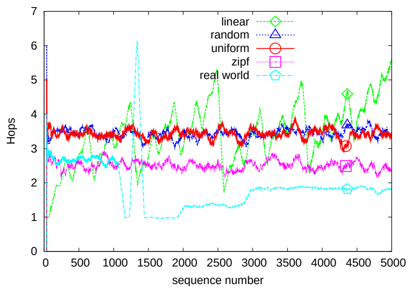
Real World Set The loaded Real World set has a very wide association tree with a small depth, except for one very deep branch. A random agent policy for this reason has an effect of on average one hop on top of the lazy policy because of the jump back to the origin, see Figure 6.14. Because the Real World set has a value range that is much smaller than the other sets, querying it with those quickly yields in out of scope value retrieval. The effect of this is that the one box that is responsible for the last range that reaches to theoretical infinity is queried for all these values. This can very well be observed through the Linear set using the lazy policy. After around 1700 queries, the hop count does not change any more, indicating the value range has been exceeded. The Linear set also shows that most of the lower values are found on the same depth in the association tree, most probably on the same site. In between a peak is found where the single deep branch is being followed. Obviously, the sticky agent policy effectively removes most of the hops for the Linear set here as the last visited site is in most cases the target for the next query. The Real World set when being queried shows how the values are being distributed over the set. The peak that the Linear set encounters is a result of the values at the end of



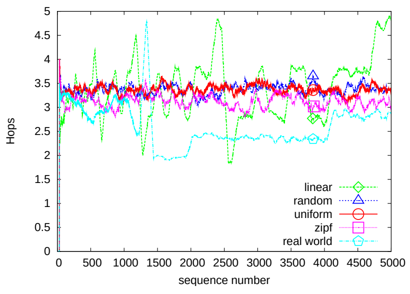
(a) Lazy policy



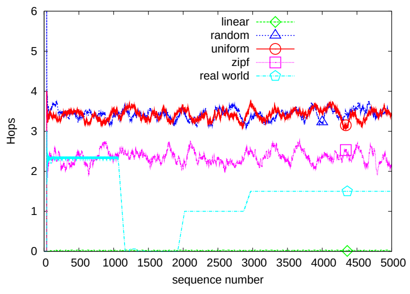
(a) Lazy policy



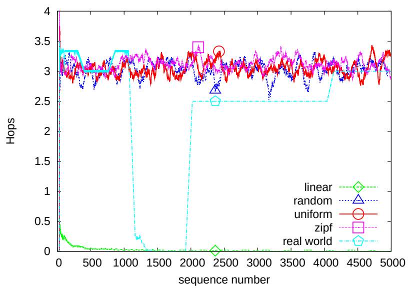
(b) Random policy



(b) Random policy



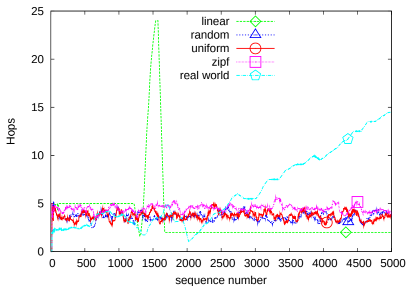
(c) Sticky policy



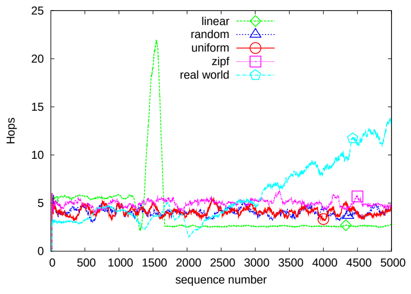
(c) Sticky policy

Figure 6.12: Querying the Uniform set.

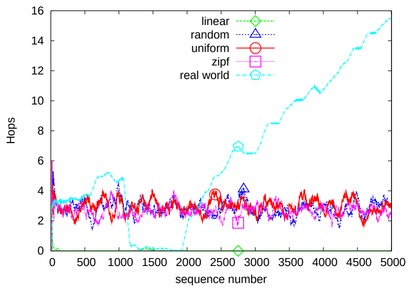
Figure 6.13: Querying the Zipf set.



(a) Lazy policy



(b) Random policy



(c) Sticky policy

Figure 6.14: Querying the Real World set.

the Real World set. Note that the hop counts for the Real World set appear to be not as high as the peak of the Linear set. This is an effect of the moving averages being compared, where the Real World set apparently has alternating values causing high and low hop counts. The sticky agent policy helps to reduce the hop counts in the small linear part of the Real World set.

Cache Policy From the loading and querying simulations we can conclude that there is no single agent policy that suits best for all cases. Without any help from the Armada cluster, the lazy policy achieves the lowest hop counts for any random based set, ignoring the cache policy. The sticky policy only performs well on a set that makes the agent often visit the same site in succession. This typically happens in a linear sequence or stable value case. The sticky policy can be considered to be a limited cache policy. It stores at most one location, but does not use the information present therein and always unconditionally returns to this location stored in the "cache".

The cache policy which we have mostly ignored before, outperforms any other policy by far. Its superior low hop counts are mainly due to the unlimited amount of cache slots which eventually allow to collect all trails available in the entire Armada. Mainly because of this unrealistically high (and theoretically unbounded) storage capacity, this policy in its current form is considered to be artificial and only feasible in a hypothetical world. The more trails are stored, the longer the time it takes to search through these trails. Since trails are only appended, this just makes the cache lookup slower and slower over time. The problem is made worse given that each trail has to be searched step by step to find a possible best match from the cache. However, its supreme performance win cannot be ignored. To be able to understand this performance and possibly approach it with a much more realistic policy, we have to look in more detail into the association tree and in particular where most of our hops go.

The Random sets are a good starting point for this performance quest. They show a very stable average of hop counts for all policies, where the lazy policy performs slightly better. The reason for this, is the price one has to pay for an association tree branch mis-prediction, which leads to a jump back – in the worst case to the origin. Such jump back to the origin, the lazy policy never has to make, since it always starts there. The cache policy performs so well on the same set, simply because it hardly mis-predicts. Because it considers its own cache, it always knows the origin, resulting in an equal to lazy performance in the worst case. However, if there is a cache item for the right branch, the cache policy can use it, jumping ahead in the right direction. The more trails cached,

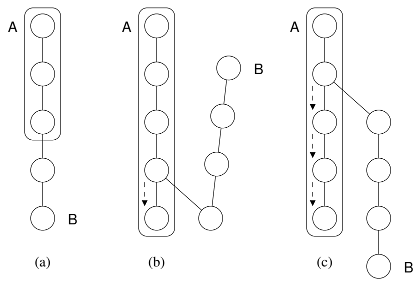


Figure 6.15: Association trial intersections.

the more precise the cache policy becomes, which eventually means that the chosen site for a query is immediately the right one.

6.8 Cache Policies

Our random agent policy simply does not take any branches into account. If it ends up in the right branch, then this is pure luck. The sticky policy only ends up in the right branch if the workload allows for this, as mentioned before. The cache policy gets most of its performance from jumping in the right branch of the tree. Hence, an agent would greatly benefit from having a cache which contains a number of trails for separate branches, which are taken as starting point. Experiments are necessary to show the trade-off of storing those trails against the gained performance. A brute force policy to just store the last x different trails would help, but probably needlessly store a lot of duplicate data. As each trail includes the full parent trail most of the information may overlap. It would be interesting to try and detect this overlap, and to store the trail that has the most depth. Problem here is to decide when it is or is not paying off to discard a previous trail in favour of a newer one. The common part of both trails may be small, and hence resulting in loss of branch information.

Cache Metric A metric that we can use here is the length of the trails after their common part starting from the origin. Consider Figure 6.15 depicting three situations where two trails intersect. In the figure, only the sites referenced in the trails are depicted. This equals the association tree, and hence can have a situation as in Figure 6.15(a) where trail $A \in B$. Obviously, for this situation, trail *B* can be chosen without losing any information, as we can reach the same sites as before. As a metric, for this situation we can define that by replacing *B* with *A*, we reduce the possible hops we have to take for any query at maximum by 2 hops. At the same time, we do not add any additional hops in the worst case scenario, as all sites from *A* are contained in *B*. Figure 6.15(b) on the other hand shows trail *B* which is much more specific than *A*, but does not fully contain *A*. In the depicted case, it may be evident that the loss of discarding trail *A* does not outweigh the win of storing trail *B*. The to be discarded site from *A* can be reached via *B* by stepping from the last site in the common part of both trails. In terms of hops, this case reduces the number of hops at maximum by 4, while it increases them at maximum by one. Lastly Figure 6.15(c) shows trail *A* and *B* where the overlap is partial and the benefit of either over the other is not obviously clear. Applying our metric, the maximum number of hops is decreased by 4, increased with 3. Though the loss of either branch is substantial. It may be clear that when the cache slots are all filled, an algorithm to find which trail should be dropped — if any — needs to be run. From the metric used before, we can define the benefit ratio as the maximum number of reduced hops divided by the maximum number of added hops. This ratio has a value greater than 1 for trail *A* against *B* if *B* reduces more hops, than those lost by removing *A*. When the ratio is smaller than 1, trail *A* is favourable for the system as a whole. When there is no loss such as in Figure 6.15(a), the ratio cannot be computed. This is not a problem, as in such case *A* can always be replaced by *B*. A cache insertion algorithm that makes use of this ratio is depicted in Algorithms 6.1 and 6.2.

Figure 6.16 depicts the final state of the cache trails after a query run when the cache allows for 5 trails. While 5 trails are insufficient for each tree to reach every leaf node, the figure clearly points out that the available trails are not positioned in the most effective locations. In particular, a lot of redundancy is contained in the used trails.

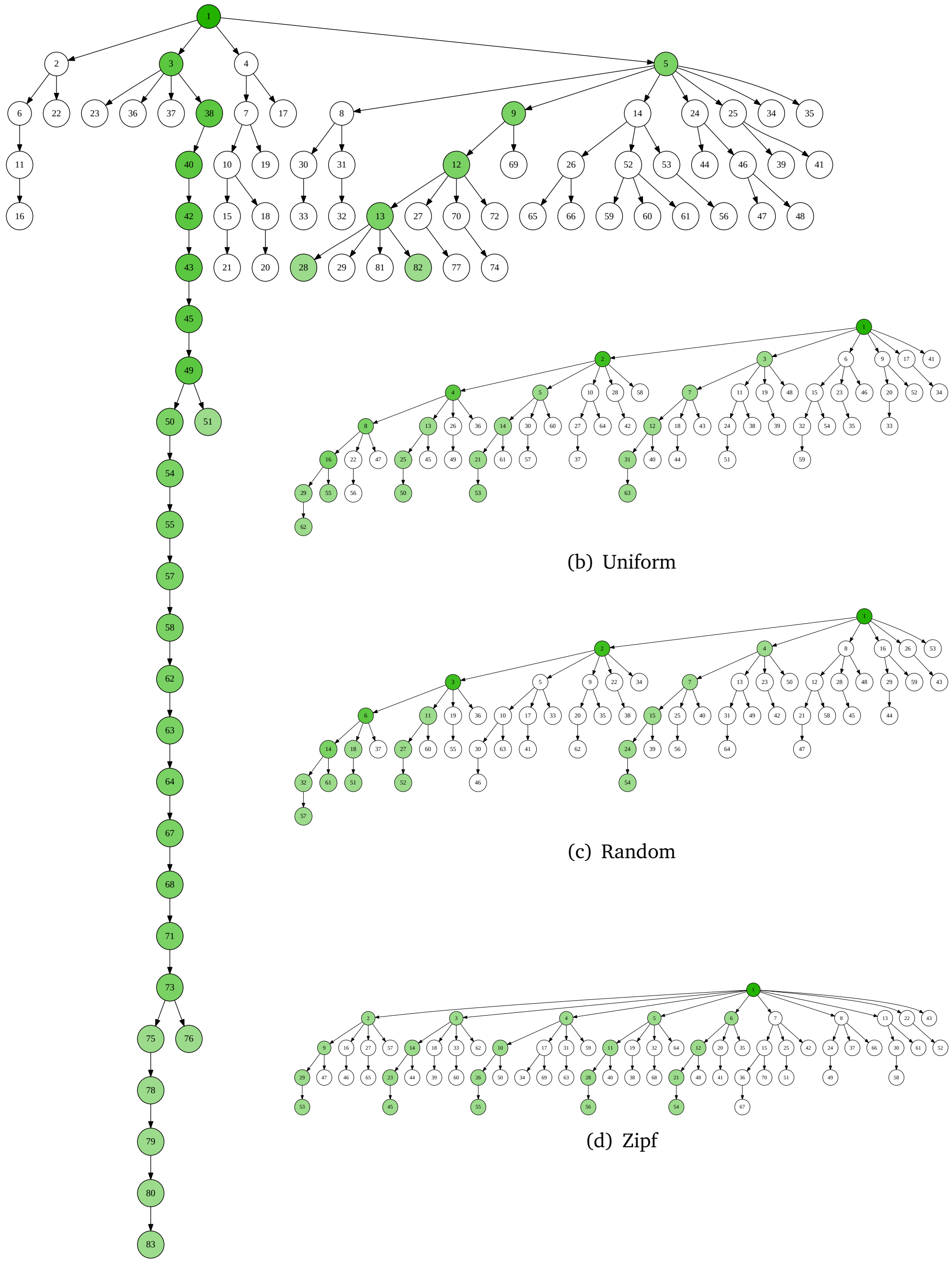
Benefit Ratio From Figure 6.17 the average number of hops taken per query for various cache sizes can be read. From the Random set in Figure 6.17(b) it immediately shows up that the performance for 1, 2 and 3 cache trails is roughly the same. The next performance wise jump is made by 4 and 5 cache trails. This behaviour can be explained by the graph from Figure 6.16(b). Obviously the leftmost (and longest) trail is always in the cache, as it is the most beneficial trail according to the benefit ratio. The algorithm adds the leftmost two next trails to the cache first, resulting in an almost equal performance. The trails have a very

Algorithm 6.1 Cache insertion algorithm.

```
$candidate \Leftarrow \emptyset$
$maxratio \Leftarrow 1$
**for each** $cache\ t$ **do**
    $ratio \Leftarrow \text{benefit}(newtrail, t)$
    **if not** $ratio$ **then**
        $\text{replace}(t, newtrail)$
        **break**
    **else if** $ratio > maxratio$ **then**
        $maxratio \Leftarrow ratio$
        $candidate \Leftarrow t$
    **end if**
**end for**
**if** $candidate \neq \emptyset$ **then**
    $\text{replace}(cache, candidate, newtrail)$
**end if**
```

Algorithm 6.2 Implementation of the `benefit` function.

```
$i \Leftarrow 1$
**while** $A_i \neq \emptyset$ **do**
    **if** $A_i \neq B_i$ **then**
        **break**
    **end if**
    $i \Leftarrow i + 1$
**end while**
**if not** $A_i$ **and not** $B_i$ **then**
    **return** $0$
**end if**
**if not** $A_i$ **then**
    **return** $\emptyset$
**end if**
$lenA = \text{len}(A) - i$
$lenB = \text{len}(B) - i$
**return** $lenB - lenA$
```



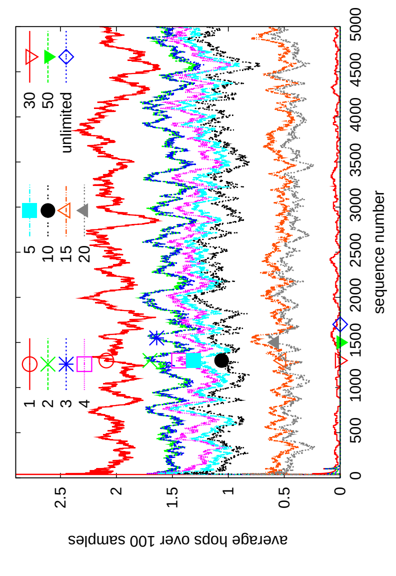
(b) Uniform

(c) Random

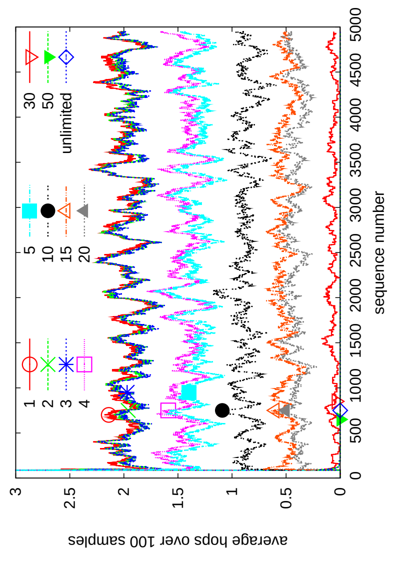
(d) Zipf

(a) Real World

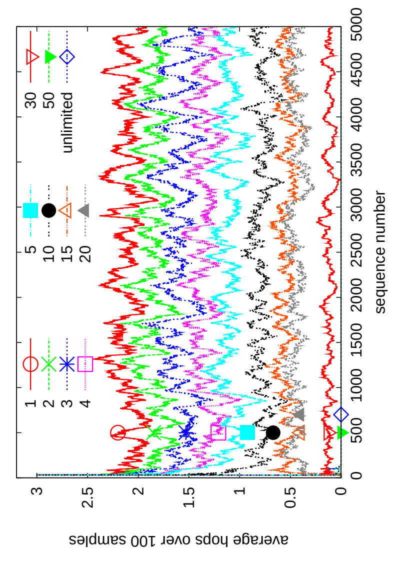
Figure 6.16: Final state of first generation cache trails after querying.



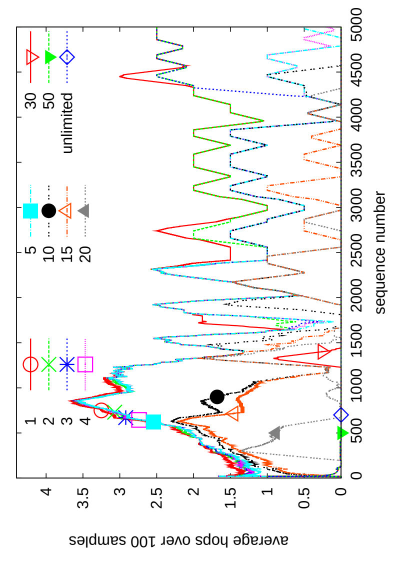
(a) Uniform



(b) Random



(c) Zipf



(d) Real World

Figure 6.17: First generation cache with hops for different cache sizes.

large common part, but are selected by the algorithm because they have a larger benefit ratio than other (shorter) trails. The next added trail is the rightmost trail from the figure, which explains the performance gap between 3 and 4 trails in the cache. Adding the fourth trail from the left in case of 5 trails in the cache results in the little performance win between 4 and 5 trails. It is obvious that the chosen trails to cache are quite inefficient for the total picture. Instead a cache utilisation such as chosen for the Zipf set in Figure 6.16(c) is much more efficient. The hops diagram for this set in Figure 6.17(c) shows an improving performance per added cache trail. The algorithm chooses better in this set because it operates on such a wide association tree, where the deepest leafs are all in a separate branch of the tree. Since the depth of most of these are equal, the algorithm does not consider a trail which is almost contained in another already cached one as better than one from another branch as happened in Figure 6.16(b).

It is obvious that the algorithm in its current form is not choosing the ideal trails for its cache. This is most prominently shown by Figure 6.16(d) where three trails are used for the deepest branch. The two trails used for the single site splits off of the main branch actually add very little (one hop) extra knowledge given the longest trail, considering other branches that are not in the cache, but could have been cached instead. We can conclude that with the current algorithm, the amount of overlap with other trails in the cache is ignored. This results in trails that are very close to other cached trails to be added in favour of other cached trails which have a smaller benefit ratio. The trail that is added to the cache as a result simply is a loss in the total picture of the cache coverage. The benefit ratio algorithm needs to be refined to take the overall benefit for the cache as a whole into account when replacing a cached trail for another.

Looking at the cache trail trees from Figure 6.16, it is persuading to think that siblings are directly reachable from the trails themselves. However, this is not true, as the vanilla Armada model defines not to include sibling information in the sibling trails themselves. They only have the predecessor trail, hence this information is not available. Also, because the trails are depicted on an association tree, it is hard to see that there is a temporal relation between all direct successors of the same step. This means that even if sibling information was passed onto the successors, this still would not include the full set of siblings, but only the siblings that were involved in the same chunk operation. For this reason, the cache trails as depicted in the figures, describe the full “span” of the trails. Any optimisations to the cache replacement policy need to take into account that only this information is contained in each cache trail.

Algorithm 6.3 The `cacheappend` function.

```
Require: $new \neq \emptyset$
  cacheaddifnotcontained($new$)
  if size($cache$) $>$ MAXCACHESIZE then
    removeleastfromcache()
  end if
```

Algorithm 6.4 The `cacheaddifnotcontained` function.

```
  for each $cache$ $t$ do
    if $t \subseteq new$ then
      cachereplace($t, new$)
      return
    else if $t = new$ then
      return
    else if $new \subseteq t$ then
      return
    end if
    cacheadd($new$)
  end for
```

Improved Cache Algorithms 6.3, 6.4, 6.5 and 6.6 depict an improved version of the cache replacement algorithm. Instead of comparing a new trail to each of the trails in the cache separately, the new trail is compared to the other trails in the cache as if it were part of the cache. This leads to removal of the trail in the cache that results in the least loss in terms of benefit. The essential difference between the first cache replacement algorithm and this algorithm is that the benefit is no longer calculated based on solely the trail itself. The benefit is now calculated as the number of hops that are reduced considering all other cache trails. As a result, those sites (hops) that are in common with other trails do not count for the benefit any more. For this, the longest part in common with the other trails in the cache has to be determined, to calculate how many sites are uniquely added to the list of known sites by the trail.

Algorithm 6.5 The `removeleastfromcache` function.

```
  $min_b \Leftarrow \text{inf}$
  **for each** $cache\ t$ **do**
    $b \Leftarrow \text{length}(t) - \text{commonlength}(t)$
    **if** $b < min_b$ **then**
      $min_b \Leftarrow b$
      $cand = t$
    **end if**
  **end for**
  $\text{cacheremove}(cand)$
```

Algorithm 6.6 The `commonlength` function.

```
**Require:** $t_a \neq \emptyset$
  $max_s \Leftarrow 0$
  **for each** $cache\ t_c$ **do**
    **if** $t_a = t_c$ **then**
      **continue**
    **end if**
    $s \Leftarrow \text{commonpart}(t_c, t_a)$
    **if** $s > max_s$ **then**
      $max_s \Leftarrow s$
    **end if**
  **end for**
  **return** $max_s$
```

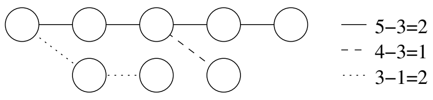


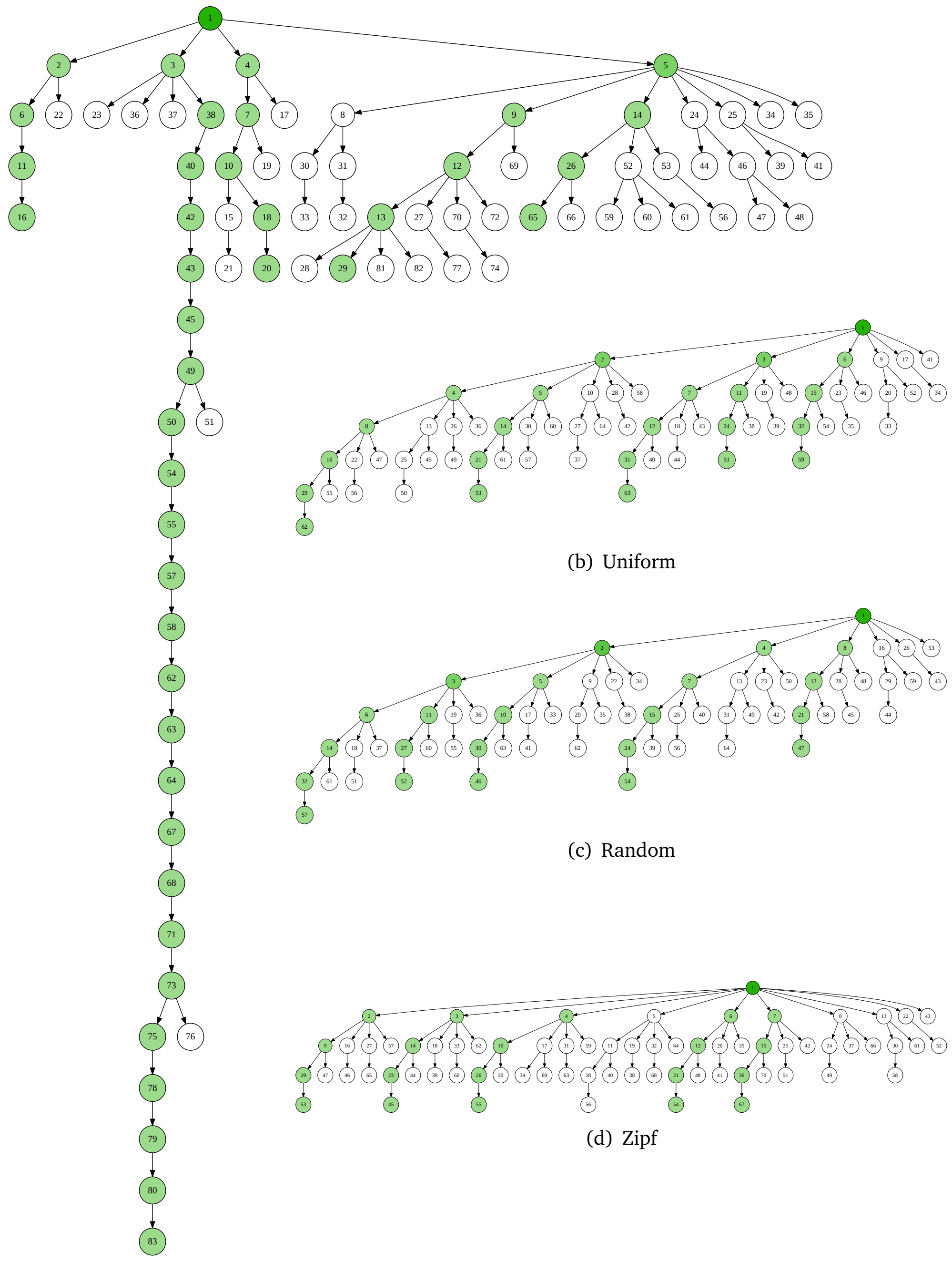
Figure 6.18: Benefit calculations for three trails.

The cache replacement algorithm works by requiring one extra slot in the cache to store a new trail. To ease the algorithm, a new trail is only added if it is not already in the cache, or superseded by a trail from the cache. Also, when a trail is found that supersedes a trail from the cache, it is used as replacement for the superseded cache trail immediately. This way, trails added to the cache are always trails that address a site which is not addressed by all others.

If the number of trails in the cache exceeds the maximum number of allowed trails, the cache replacement algorithm is run to evict one trail from the cache to be removed. The trail to be removed is chosen based on the afore described benefit function. For each trail in the cache, the benefit is calculated, and the trail with the smallest benefit is chosen to be removed. Figure 6.18 depicts a situation of three trails. On the right of the picture the benefit calculation for each of the trails is shown by taking the total length subtracted by the length of the part of the trail in common. It is obvious that the trail with benefit 1 would be evicted in favour of the other two with both a benefit of 2. Note that after removing this trail, the benefits of the other two trails have to be recalculated because the common parts may have changed, as is the case for the longest trail in the figure.

It is to be expected that there is not always a single trail that has the lowest benefit. There may very well be multiple trails matching. The algorithm removes the oldest trail in such case, as it is based on a cache that is implemented as a linked list, where new trails are appended to the tail. Hence, the first trail found when traversing the list is the oldest. The rationale for doing this is that the more recently added trails may better reflect the current query behaviour.

Gradual Improvement From Figure 6.19 it can be deduced that the second generation cache trails replacement policy reaches a better final state of cached trails. Compared to Figure 6.16 more branches are represented in the cache, and trails with large overlapping parts are no longer present. This is entirely conform the objectives of the improved replacement algorithm. The performance wise results are depicted in Figure 6.20. When compared to Figure 6.17 we observe that with 10 cache trails the second generation has an average per-



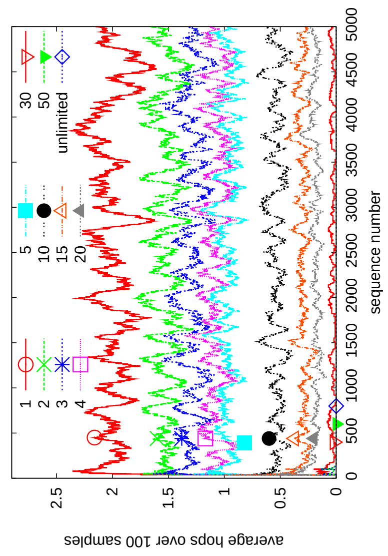
(b) Uniform

(c) Random

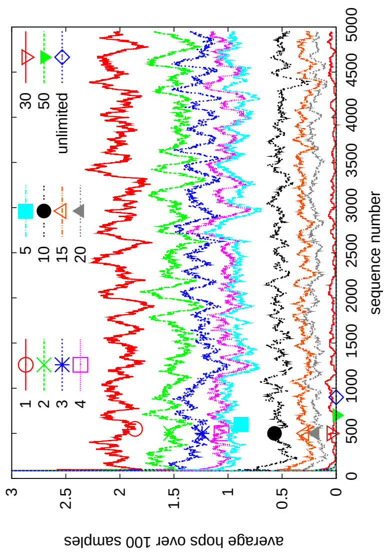
(d) Zipf

(a) Real World

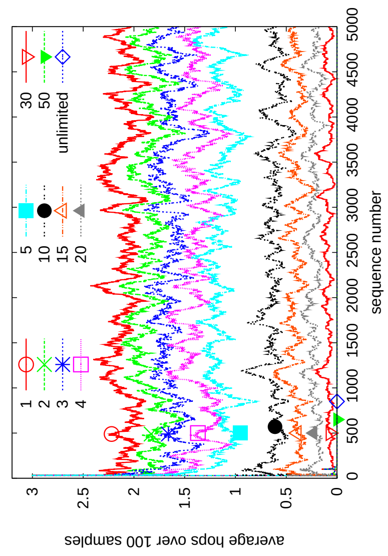
Figure 6.19: Final state of second generation cache trails after querying.



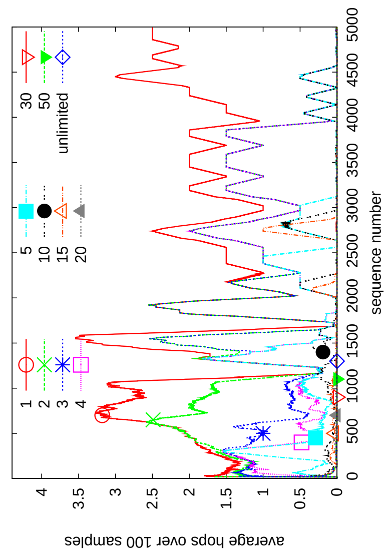
(a) Uniform



(b) Random



(c) Zipf



(d) Real World

Figure 6.20: Second generation cache with hops for different cache sizes.

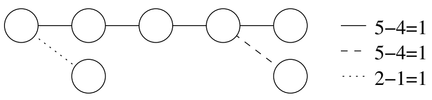


Figure 6.21: Calculations for three trails with equal benefits.

formance around half a hop per query, whereas the first generation delivered an average performance of around 1 hop per query for the random sets. Unlike the first generation, the second generation shows a more gradual performance improvement for the Random and Uniform sets per added trail to the cache size. The Real World set in general has an improved performance with the second generation cache replacement policy. However, for the cache size of one trail, a peak can be seen for the second generation that is 1 hop higher than in the first generation. The peak is caused by a linear query pattern, which values are found in the long trail of the Real World association tree. The hops are caused by the values that are stored in the start of the long trail. Once the values address a site that has a longer trail than those stored in the cache, it is being added, and hence the hops immediately drop to a minimum, as successive values at need one hop every time the site capacity has exceeded, like in the linear case is happening. The single hop is explained by the benefit function disregarding the common part of trails in the second generation. For this reason it takes one site longer before the benefit is higher since the root node is always shared with other trails.

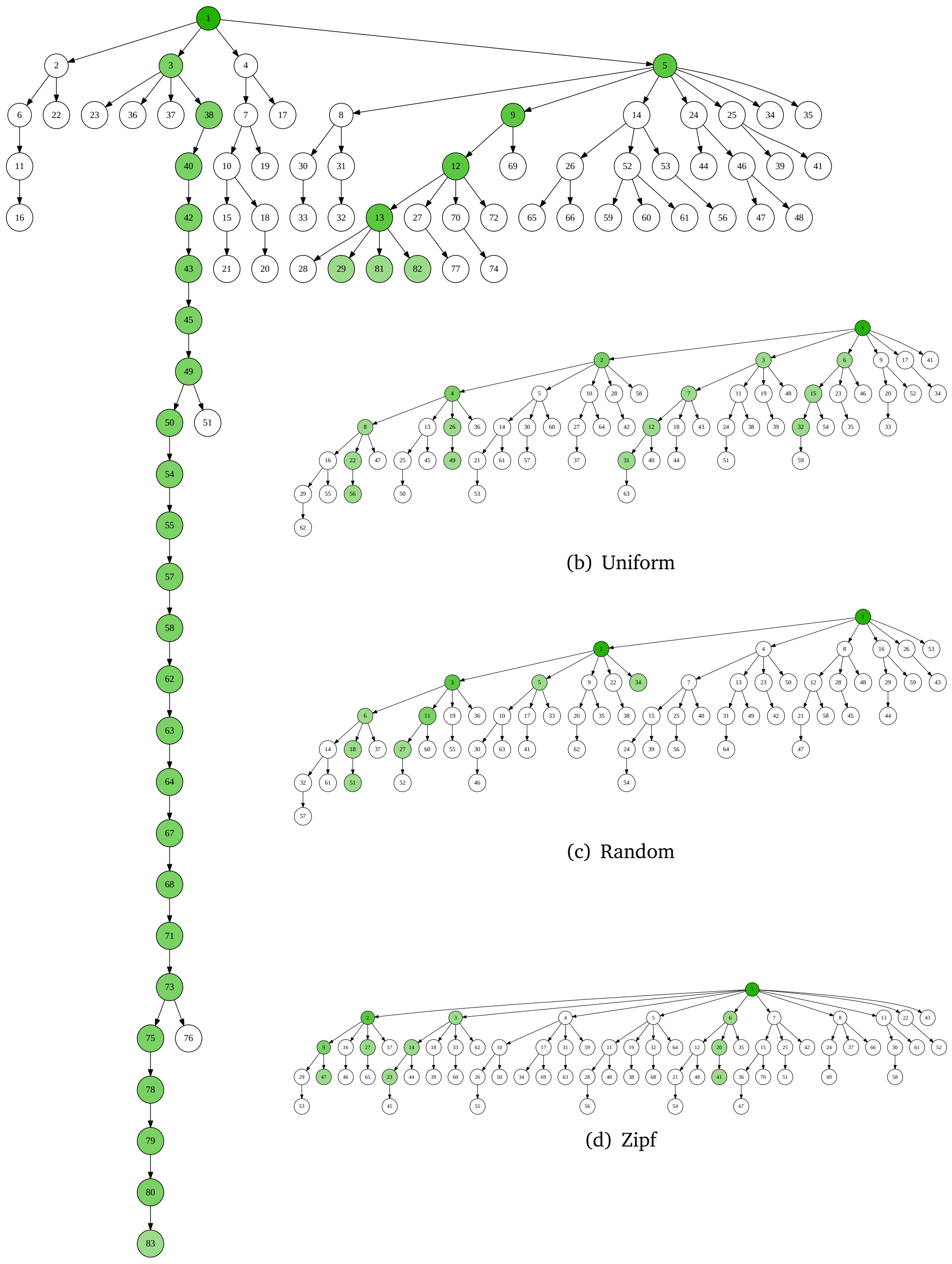
Wide Coverage When observing the trail transitions in the cache using the second generation cache replacement policy, situations similar to Figure 6.21 appeared to be common practice. What happens is that the dashed trail is added to the cache, while the dotted trail is the oldest. All trails have an equal benefit according to the second generation cache replacement policy, and hence its default method of removing the oldest trail results in the two longest trail to be kept, and the dotted trail to be removed. While in absolute terms all three trails add the same to the system as whole, the dotted trail can be more useful to the system considering heuristics. First sites up in the tree have a higher chance of having many (direct and indirect) successors over sites lower in the tree. For this reason, sites up in the tree possibly help for more queries, due to their potential coverage. Second, sites up in the tree have a higher chance of addressing a not yet addressed branch of the tree. As we have discussed previously, getting distinguishing branches in the cache increases the theoretical coverage on the tree. Because of these heuristic hypothesises we refined the second algorithm for the cases where an eviction decision is made for trails with an equal benefit. Instead of choosing the oldest in the cache, an evaluation is made aimed at retaining the trails targeting sites higher in the tree.

A few variants are possible to achieve the aforementioned goal. First the algorithm could consider the common part of trails and prefer trails with a lower common part as this means they distinguish themselves better from other trails, than those with a large common part. Naturally, this favours short trails, as they have a small common part given that the benefit of the trails being compared to is the same. Short trails obviously address sites higher up in the tree. Secondly a relative benefit based on the depth of the trail could be calculated. In the given example from Figure 6.21 the dotted trail would have a benefit of 1 weighted over a common trail of length 1. The dashed and normal trail have a benefit of 1 over 4, resulting in one fourth. Alternatively, the benefit is made relative to the depth in the tree where the non common part of the trail starts. Effectively, the same node is selected to be retained by both variants. Tests have indicated no performance improvements compared to the previous generation. Since this approach is a micro optimisation that only affects a few cases, this is not a surprising outcome.

6.9 LRU Cache

Until now we have only considered caches based on the Armada lineage trail structure. As a result our caches were focused on keeping the optimal trails for the tree structure. The behaviour of the clients per their queries has been ignored. Patterns of interest in only a specific part of the Armada are not recognised by the till now implemented cache strategies. From observations of the cache transitions, strategies that adapt to the current workload may improve performance in cases where sibling nodes are alternatively selected to be included in the cache. Those cases clearly indicate a localised query interest, but cache trails are assigned for those sibling nodes, since other (longer) trails are considered to be more valuable for the entire tree and any possible query.

We implemented a simple LRU cache strategy for Armada trails. Without assuming any knowledge of trails, the LRU simply stores the trails that point to the box for each query. Trails that already exist in the cache are removed and reinserted. When the cache overflows, the trail that is least recently inserted is removed.



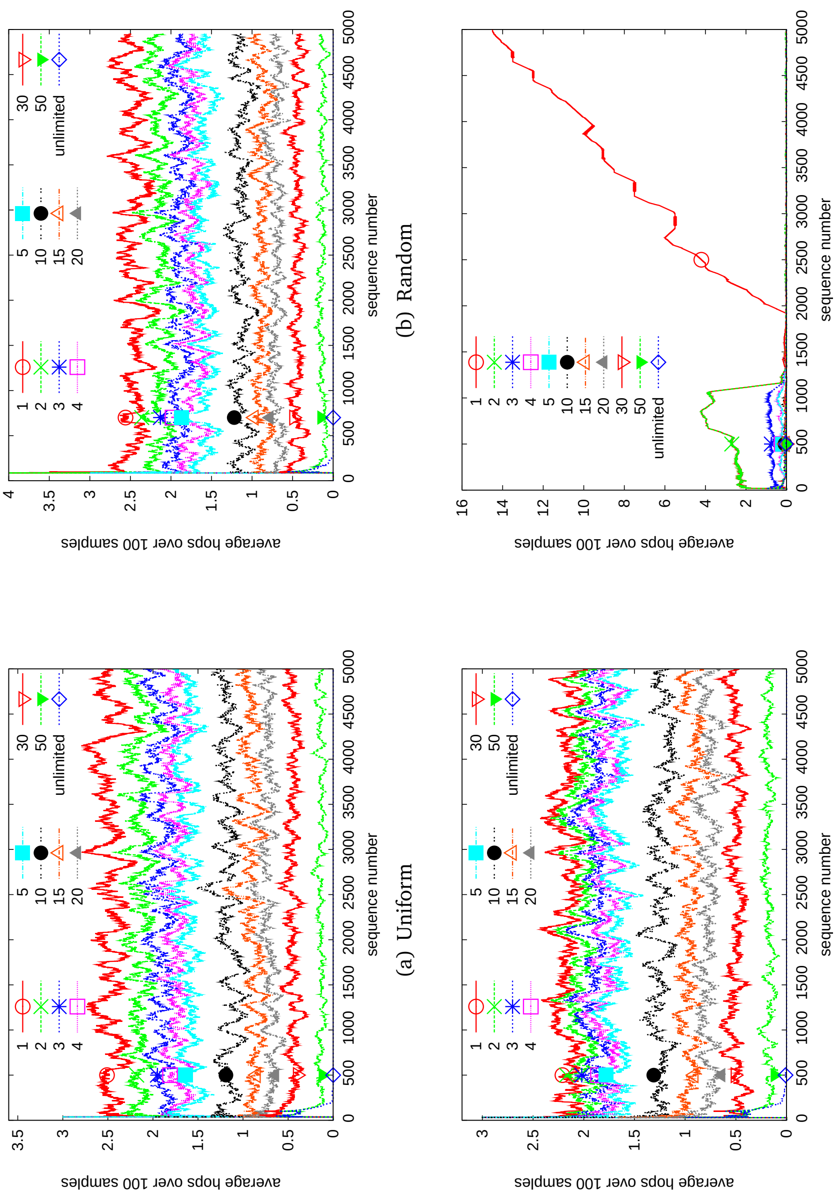
(b) Uniform

(c) Random

(d) Zipf

(a) Real World

Figure 6.22: Final state of first generation LRU cache trails after querying.



(a) Uniform

(b) Random

(c) Zipf

(d) Real World

Figure 6.23: First generation LRU cache with hops for different cache sizes.

Figure 6.23 shows the average hopcount graphs for this LRU cache replacement policy. Again the graph for the Linear set is omitted, as it is a straight line close to zero hops on average. The LRU is obviously very well capable of "following" the linear case. The random sets show a steady and slightly worse performance compared to the first non-LRU cache generation. In all cases adding a cache trail results in a reduction of the average hop counts. More interesting is the Real World set. It shows for more than one cache trail that the entire tail starting from around the 1900th query can be reached in close to zero hops. This is very well explained because that entire tail consists of an alternating low/high value sequence. Starting from two trails, the LRU can keep the low and the high value sites in its cache and hence serve the query load very well. Starting from 3 trails, the entire Real World set can be done with on average less than 1 hop, which is quite effective.

Overlap Awareness Because the LRU cache does not take anything into account from the Armada model, but just stores trails, the chosen trails are not optimal considering the entire path up to the root is used by the agent when using the cache. From Figure 6.22 this particular problem can be observed, as only 4 trails can be seen, while there are 5 in the cache. In all four cases there is a trail that is fully contained in another in the cache, hence invisible in the figures. Like we do in the non-LRU cache policies, the LRU could be extended to recognise when trails are contained in each other and then move the largest trail. When a site is found in the predecessors of a trail that matches, the trail for that site is added to the LRU. Hence making the LRU aware of this, can improve the effectiveness of the cache trails.

Experiments show that applying above strategy indeed improves the effectiveness of the cache trails. For all sets, a cache size of 50 is now sufficient to reach a near zero hops performance where the previous generation was not able to achieve that. Compared to the third generation non-LRU cache, the second generation LRU cache is roughly half a hop worse in performance for the random sets. The performance on the Real World set has not improved over the previous generation, but still outperforms non-LRU caches.

Cache Performance Comparisons We have considered the agent's query behaviour using a least recently used scheme, where the last used trails are kept in the cache. This cache differs from the non-LRU caches in that it is fully driven by the query load, possibly adapting to the query points of interest. However, this strategy on average performs worse than non-LRU, trail logic based caches for all but one set. The cache replacement based on LRU may be too sensitive to outliers and hence discard trails too often. An alternative here to adapt to the query workload, is to use the usage rate of trails in the cache. This usage could be either defined as the number of times the trail is used, or as the number of times the sites contained in the trail are used. However, for such a strategy it is necessary to keep the usage count for every trail or site, since otherwise trails will never be added to the cache as they are never more used than those in the cache. Keeping usage information about all possible trails or sites is as roughly the same as having an unlimited cache, and hence not a viable solution.

Summary

The Armada agents have to locate data in the system. They do so by following lineage trail information, available on every site. An Armada that has grown large involves many sites, which all potentially can contain the data an agent is looking for. While network connections are expensive, time wise, the more an agent needs to hop around, the worse the performance.

Four agent policies have been studied to see the effect of them on five data sets. While different sets result in trees of different depths, the hops taken by an agent are affected by this depth. While some policies work reasonably well on some sets, only the policy where the agent caches trails for later reuse reaches a very good performance for all sets.

Since an unlimited cache is a rather unlimited resource claim, we conducted several experiments with limited cache sizes. By revising our cache algorithms, based on characteristics of Armada lineage trails, we reached an acceptable amount of hops per query for a limited amount of cache. This result indicates that the active Armada client is viable in terms of costs with respect to the autonomy and distribution it allows.

7 SQL Approach

7.1 An Implementation's Architecture

In an Armada cluster, querying is done by an Armada agent. This agent bounces back and forth between nodes of the Armada when resolving queries. This way, the agent relieves the nodes from performing query tasks and interacting with their co-nodes. An important aspect of the agent is that it keeps track of the lineage trail information that it encounters during query traversals through the Armada. This particular behaviour allows the agent to start new queries on a node that suits the best based on its own knowledge. By doing this, an agent avoids needlessly querying nodes that cannot answer its question. In addition this implements a strategy to bypass the origin node in most cases, effectuating hot-spot avoidance.

Workers SQL systems currently do not have any means of providing feedback with the user. This naturally rules out the interaction part of the Armada that needs to go beyond the agent. Conceptually, Armada agents are not related to the data hosting nodes in the system. Each agent comes in the form of a special equipped database system, and needs to run on a node. Each node in the Armada may offer agent functionality next to data hosting, but this is not required. In case both agent and data share the same node, the lineage trail information between the two can be shared to increase both their knowledge. Agents interact with Armada nodes to execute queries, on behalf of a client. While agents need to be able to find the Armada nodes, clients need to be able to find an agent in order to execute a query. For agents it is sufficient to know one node of the Armada, the lineage trails point the agent to the other nodes it needs. However, for a client, such redirection is not in place. Agents can work in a pool of "workers", where a client just gets one assigned to perform a query.

Here it is not important which agent is being assigned, and the pool may grow or shrink based on the number of client requests.

Starting Point The problem yet remains how a client can get to an agent in the system, willing to do the work. There are ample opportunities here, e.g. using a DHT or an Armada structure. However, eventually they all need a starting point. Either a server that can tell what key to look for, or a first server that can point into the right direction. Hence, we assume that we just have a central server that assigns an agent to a client via a simple redirection a job that is simple enough not to become the actual bottle-neck of the system. The central locator service, maintains a pool of all active agents. A client connects to the locator and gets a redirect to an agent that handles the client's query. This has the advantage over e.g. DNS-based load balancing that the pool of agents can change without information getting stale, such as out of date DNS entries. In addition, statistics can be collected to be used to control the agent pool size, because all clients first pass through the locator. Last, the locator could track the load of the agents and try to assign new clients to the least loaded agents or another scheme which allows for better load control than randomisation.

Catalog So far we have assumed that clients connect to agents, which in their turn know how to deal with the entire "database" that is hosted in the cluster. In reality, an Armada represents at most a table, but maybe even only a column of such table. As result, many Armadas are present in the cluster, and each agent needs to know about them, to be able to query them. Next to multiple Armadas being present, there is also a relation between most of them. Several columns form a table, some tables form a schema, and so on. Also this information, usually referred to as a "catalog", is necessary for each agent to function properly. The catalog contains for each Armada at least one node that that hosts that particular Armada — typically its origin — and all relations between the Armadas. Note that growth operations applied to each individual Armada need no modifications to the catalog. The catalog only needs to be modified, when new Armadas are added, old ones removed, et-cetera. Given that such modifications to this catalog are rare, it can be stored centrally, without performance penalties. An already central place in the system is the locator, which assigns an agent to each client request. Storing the catalog in the locator, allows each agent to retrieve catalog information by contacting the locator. However, the locator may soon get overloaded by the many requests of the agents, for this relatively static data. Hence, each agent can replicate the catalog for its own private use. Here a consistency problem arises, but given the nature of infrequent updates to the catalog, a simple pushed based update from the locator to the agents is sufficient.

For each agent that is created to serve in the system, an initiation ritual has to be performed, such that the agent can start serving client queries. This ritual includes making itself known to the locator and retrieving catalog information such that queries can be resolved. The catalog being received is simply a copy from the locator's catalog. After this has been set up, the agent is registered with the locator, which can start assigning clients to the agent. Note that the catalog does not include the state of each Armada being referenced. The agent starts with a clean record, and as such its first queries are not well targeted. However, along the way the agent soon builds an internal representation of the Armadas it deals with in terms of cached trails, allowing for a smaller data search time. Once an update is made to the catalog, the locator is being updated. It functions as master “database” for all agents as “replicas”. The locator can send out direct updates for all the agents. However, this generates a direct load to perform all these updates, and may on the agent side conflict with pending actions. Hence, a simple delayed update strategy can be used by the locator. Once the locator assigns a client to an agent, it first sends all pending updates to the agent. Since these updates are small and fast operations the delay for the client is negligible while it performs an update of the agent just at the moment when it is necessary, keeping the catalog for the client unmodified — i.e. no transaction aborts due to catalog updates.

Using this scheme, existing agents can drop out by simply denying connections from the locator. Once the locator notices that an agent no longer responds, or appears to be entirely missing, it simply unregisters it from the pool. From that point the agent is not considered to be an agent any more, and since it is no longer in the pool, it is not assigned any client requests. In fact, the locator is not aware of its existence any more. If said agent becomes available again, it simply has to register itself again with the locator, as if it were a new agent. This means its catalog is replaced with a fresh copy from the locator. The cached lineage trails that the agent still has can be kept, as the information stored in them is never incorrect; at most out of date. This can give a reconciling agent a jump start due to its previously acquired knowledge about the Armadas.

Catalog Updates An update to the catalog is typically made by a client query. An operation such as creating a new table, and hence a new Armada, needs to be stored in the catalog. If the agent dealing with such operation stores the modification in its own catalog, conflicts may arise, and other agents are not able to see the new table. Hence, updates to the catalog need to be made on the locator, and immediately synced with the agent performing the update, such that it can continue processing its client's queries. In detail, adding a table or column requires new Armadas for those to be created. When such update request arrives at the locator, it creates new Armadas as necessary on nodes in the system, and stores those in its catalog. This procedure of finding a suitable node in the system is again supported by the locator in the system.

Performing Operations For each chunk or clone operation and for each Armada creation, a new node in the system has to be found. The locator maintains next to a pool of known agents, a pool of data nodes. Once a node wants to perform a chunk or clone operation, or an agent requested a new Armada to be created, the locator picks a node from its data pool. The strategies for picking a node from the pool may be based on simple heuristics such as available resources or usage statistics. Alternatively bidding procedures can take place to pick a site from the pool. In any case, the locator only maintains the pool of nodes and sends messages to each node to obtain information about their current state. Once a node is found, a new box is created on the node, and its trail stored in the catalog, effectuating the operation.

Creating a new agent in this system, is a relatively cheap operation. Basically only the catalog needs to be transferred, and the agent is ready. This makes the loss of an agent only a problem with regard to the available agents in the pool and their workload. Since the state (and involved nodes) of each Armada is not recorded in the catalog, all this information needs not to be copied. This is an advantage over systems where all information about all participating nodes is being kept in the catalog, as this requires much more data copying, as well as keeping that data consistent between the copies.

7.2 SQL Armada

The locator architecture lays a foundation for essentially getting a client to perform a query on an Armada. With SQL being the *de-facto* query language of database systems, we studied the feasibility to implement the Armada model in an SQL database. Mapping the basic metadata administration and structural operations of the Armada model to standard SQL allows us to strive after a *minimal-invasive* Armada-implementation, i.e. instead of pushing the Armada functionality inside existing (or new) SQL processors, we simply add it on-top. This approach does not only allow for a system-independent Armada that could even become a heterogeneous system using different DBMSs at different nodes/sites, but also helps to leverage mature DBMSs technology for local query processing.

7.2.1 Simple example

In Figure 7.1 the ideas of the Armada model are graphically represented. Rightmost in the picture, the lineage tree of all boxes is shown. From those boxes, only three are *active*, meaning they have data. Those boxes' data are depicted in the middle bar, which represents the full table data, depicted on the left in the figure. As can be seen, the union of all active boxes' data results in the original table again. In our mapping to SQL, we use this particular observation. From

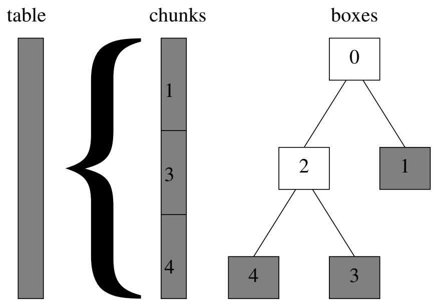


Figure 7.1: Sample Armada with 5 boxes.

the lineage trails in Figure 7.2 we can see that boxes B_0 and B_2 are not active any more, as they have successors. Boxes B_1, B_3 and B_4 contain the actual data in the Armada. Furthermore, B_0, B_2 and B_4 are positioned on the same site, S_1.

An Armada such as in Figure 7.1 can emerge as an evolutionary process over time. Each Armada starts as a single origin box, which upon need is chunked. Thereby, new sites can host boxes, to extend the Armada in its evolution. In the example, box B_0 and B_2 on site S_1 were chunked over time to make room for new data.

$$\begin{array}{lrl}
T_o = & [\,\%,\, S_1]{:}B_o\,; & \left\{ \begin{array}{l} [\,f_1,\, S_2]{:}B_1 \\ [\,f_1',\, S_1]{:}B_2 \end{array} \right. \\
T_1 = & T_o\ .\ [\,f_1,\, S_2]{:}B_1\,; & \\
T_2 = & T_o\ .\ [\,f_1',\, S_1]{:}B_2\,; & \left\{ \begin{array}{l} [\,f_2,\, S_3]{:}B_3 \\ [\,f_2',\, S_1]{:}B_4 \end{array} \right. \\
T_3 = & T_1\ .\ [\,f_2,\, S_3]{:}B_3\,; & \\
T_4 = & T_1\ .\ [\,f_2',\, S_1]{:}B_4\,; &
\end{array}$$

Figure 7.2: Sample Armada lineage trails.

7.3 Armada Evolution

In a typical database environment, data is available in *tables*. These database *objects* are subject to modifications, in particular growth. The Armada model supports these modifications in the case of growth by splitting a table into multiple tables. This can happen when this is required, and Armada keeps the administration. An important aspect within the Armada model here is that each site is considered autonomous. This means that such split is a decision in agreement with the sites involved in the split, and can be initiated by the site hosting the data.

This autonomy reflects in our view on the role of the client within an Armada. Unlike conventional clients, in Armada a client is *active*. This means for a client that it plays a central role in query execution. Instead of sites contacting other sites during query execution, the active client bounces between the sites.

Due to the autonomy of the sites, they mainly manage themselves. A site can decide that it needs to chunk its data when resources get scarce. Or, when the load is high, to clone its data. These operations can happen whenever the need arises to do so, without intervention of a client as long as sites are available and willing to co-operate.

Table Chunking Simply put, the Armada model allows to break up a table in two (or more) sub tables. This split is done based on a *chunk function* that specifies what part of the original data goes into which of the new tables. After a break up of a table, two new tables are produced and the original table is removed. References to the original table, be it in the system itself, or by users, do not work any more after this operation. To solve this, one of the new tables can be renamed to the original table. This does solve the table from being unavailable, but introduces confusion as the new table obviously doesn't hold the same data as the original one. Alternatively, all references to the original table can be updated to point to the new ones. This might be hard to achieve (e.g. humans remembering the original name) and undesirable, as now two tables have to be used instead of the original one.

Finally, to overcome the problems imposed by the previous two solutions, a special object can be provided to indicate that the original table was broken up in two. The function of this object is to be both transparent and purely informative about the break up. The object itself simply represents the *union* of the two sub tables, like the inactive box in the Armada model. It is quite naturally covered in SQL by means of *views*.

7.4 Seamless Armada-SQL Integration

With the design for a self-managed distributed Armada system at hand, we now turn our attention towards its implementation. Striving after a minimal-invasive and widely portable solution, we now describe how to map the basic metadata administration and structural operations of the Armada model to standard SQL.

7.4.1 Chunking

Each Armada starts at a single site, as a single table. In the SQL world this is an ordinary relation, e.g.:

```
CREATE TABLE treasures (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_bag_pkey PRIMARY KEY (bag)
);
```

This simple table keeps track of some treasures on site S_0. Each bag has a number which is unique, as defined by the *primary key*. The Armada model defines that this table can be broken up in two or more new tables. This has the effect of the data being spread over these new tables, which represent the boxes from the Armada model. Breaking up the original table in two, is done as follows.

First the new tables (boxes) have to be created. The creation is followed by insertion of data from the original table. The creation of the box is like the schema of the original table using the following template:

```
CREATE TABLE {armada}_B{boxid} (
    {columndefinitions},
    CONSTRAINT {armada}_B{boxid}_{pkey}_pkey
        PRIMARY KEY {pkey},
    CONSTRAINT {armada}_B{boxid}_{pkey}_check
        CHECK ({if:contrachunk}NOT{fi}
            {armada}_F{funcid}({pkey}))
);
```

In this template, variable parts are wrapped between `{` and `}`. The new tables are named after the original table, `{armada}`. To distinguish them they have a trailing `_B{boxid}` which includes the unique id of the box (table). In the template, an extra `CHECK` constraint is added that checks the primary key `{pkey}` on validity for the box. A key is not valid, when the used chunk function `{funcid}` does not hold. Two new tables are created: one that holds the data that matches the chunk function and one that holds the data that does `NOT` match. The `{columndefinitions}` and `{pkey}` are simply inherited from the original table's schema. For the original treasures table, this template results in the following create statements:

```
CREATE TABLE treasures_B1 (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_B1_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B1_bag_check
        CHECK (treasures_F1(bag))
);

CREATE TABLE treasures_B2 (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_B2_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B2_bag_check
        CHECK (NOT treasures_F1(bag))
);
```

The next step is to fill the two new boxes with data from the original table, such that that table can be removed in favour of the two new ones. The following template applies:

```
SELECT INTO {armada}_B{boxid}
    SELECT {columns}
        FROM {armada}
        WHERE {if:contrachunk}NOT{fi}
            {armada}_F{funcid}({pkey});
```

Via an ordinary `SELECT INTO` statement, only the data from the original table that matches the boxes is inserted. Many fields in the template are the same as before. Because the templates used are abstract, we illustrate their functioning with an example:

```
SELECT INTO treasures_B1
    SELECT bag, name, coins
        FROM treasures
        WHERE treasures_F1(bag);

SELECT INTO treasures_B2
    SELECT bag, name, coins
        FROM treasures
        WHERE NOT treasures_F1(bag);
```

Copying over the data is trivial and includes the same functions as the tables' `CHECK` function. Since the data was copied over to the two new boxes, the original table can be dropped.

```
DROP TABLE treasures;
```

Finally, since the original table `treasures` is dropped, it disappears, as mentioned above. The special object that represents the union of the two new boxes, is implemented by a *view*:

```
CREATE VIEW {armada} AS
    SELECT {columns}
        FROM (
                SELECT {columns}
                    FROM {armada}_B{sucboxid}
                    WHERE {armada}_F{funcid}({pkey});
                UNION
                SELECT {columns}
                    FROM {armada}_B{contrasucboxid}
                    WHERE NOT {armada}_F{funcid}({pkey});
             ) AS {armada};
```

Note that no explicit typing information is included in the view: the SQL standard does not allow us to explicitly encode it, hence the database system has to get this information based on the local box referred to in the view. The variables `{sucboxid}` and `{contrasucboxid}` refer to the box ids of the created two new tables. In the example:

```
CREATE VIEW treasures AS
    SELECT bag, name, coins
        FROM (
            SELECT bag, name, coins
                FROM treasures_B1
                WHERE treasures_F1
            UNION
            SELECT bag, name, coins
                FROM treasures_B2
                WHERE NOT treasures_F1
        ) AS treasures;
```

7.4.2 Cloning and Combining

So far, we focused on the chunk operation of the Armada model. Supporting the clone and combine operations is less trivial, but conceptually possible. Consider a clone operation to take place like previously done for the chunk operation. First the two boxes are created, but unlike the chunk operation, they have the same function as their predecessor. Hence, a clone of box B_1 from the example above results in the two boxes B_3 and B_4:

```
CREATE TABLE treasures_B3 (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_B3_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B3_bag_check
        CHECK (treasures_F1(bag))
);

CREATE TABLE treasures_B4 (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_B4_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B4_bag_check
        CHECK (treasures_F1(bag))
);
```

Next, the `treasures` view has to be made available. Here it is, unlike at the chunk operation, not trivial how to express the functionality of the clone operation in SQL. Where the union of the resulting boxes of the chunk operation results in the original box, also for the clone operation the union operator yields in the same behaviour.

Recall that as defined by the SQL:92 standard and up, the `UNION` operator eliminates duplicates by default. Hence, the union of two clones results in the predecessor of the clones. While the functionality of a clone is to provide an alternative during the query execution, the union operator feels quite unnatural. However, SQL does not have any means to express the `OR` operation on data sources, as to pick an alternative. Hence, using the `UNION` operator, a semantically correct SQL view can be created:

```
CREATE VIEW treasures AS
    SELECT bag, name, coins
        FROM (
            SELECT bag, name, coins
                FROM (
                    SELECT bag, name, coins
                        FROM treasures_B3
                        WHERE treasures_F1
                    UNION
                    SELECT bag, name, coins
                        FROM treasures_B4
                        WHERE treasures_F1
                ) AS treasures_B1
                WHERE treasures_F1
            UNION
            SELECT bag, name, coins
                FROM treasures_B2
                WHERE NOT treasures_F1
        ) AS treasures;
```

In this view the execution has to query both clones and union the two (identical) answers. While this action does defeat the use of the clones, it is correct from a generic execution point of view. There is room for improvement and optimisation here, which we discuss in a follow-up paper.

Analogous to the previous discussion, combine operations also use the `UNION` operation in SQL. In particular the duplicate eliminating nature of the combine operation when applied to data overlapping boxes, is very well covered by the SQL `UNION` operator. The combine operation does not turn one box in multiple new ones, but instead merges multiple boxes into one.

Assume we merge box B_2 and B_3 from the previous examples into B_5. This effectively means a merge of a chunk and a clone of its counter chunk are being merged. Creation of box B_5 includes the logical combination of its predecessor functions:

```
CREATE TABLE treasures_B5 (
    bag     int,
    name    varchar(64),
    coins   int,
    CONSTRAINT treasures_B5_bag_pkey PRIMARY KEY (bag),
    CONSTRAINT treasures_B5_bag_check
        CHECK ((NOT treasures_F1(bag))
            AND treasures_F1(bag))
);
```

It is not hard to see that the combination of the functions used for the predecessors yields in the whole coverage as for the origin box in this case. It is more probable that this is not the case, however. Next, the creation of the `treasures` view is in contrast to the chunk and clone operations different in that it simply unions the respective views of the predecessor boxes and substitutes the predecessor boxes with the new box B_5. This action represents the combination made of the two input boxes:

```
CREATE VIEW treasures AS
    SELECT bag, name, coins
        FROM (
            SELECT bag, name, coins
                FROM treasures_B1
                WHERE treasures_F1
            UNION
            SELECT bag, name, coins
                FROM treasures_B5
                WHERE NOT treasures_F1
        ) AS treasures
    UNION
    SELECT bag, name, coins
        FROM (
            SELECT bag, name, coins
                FROM (
                    SELECT bag, name, coins
                        FROM treasures_B5
                        WHERE treasures_F1
                    UNION
                    SELECT bag, name, coins
                        FROM treasures_B4
                        WHERE treasures_F1
                ) AS treasures_B1
                WHERE treasures_F1
            UNION
            SELECT bag, name, coins
                FROM treasures_B5
                WHERE NOT treasures_F1
        ) AS treasures;
```

The resulting view is large and contains some redundancy. However, we do not optimise the view at this stage. The only action taken here is to rename the predecessor boxes in the view into the newly created box, as they are now covered by the new box by definition.

7.4.3 View inlining

With the current technique, once created views are never updated again, leading to inevitable levels of indirections when trying to resolve them. To avoid this, a simple and relatively cheap operation can be applied on the views to update their definition by *inlining* the definition of the view they (partially) point to.

Consider again the previous example, with on S_0 the original `treasures` view which is a simple selection on box B_0. At the time of the chunk operation applied to B_0, the new definition for B_0 (as in the newly created view for it) can be inlined in the `treasures` view. For this, the reference to B_0 is replaced with the sub-query for it, resulting in the same view as on S_1 or S_2. In a similar way, inlining during the chunk operation in the transparency view on S_2, results in one level of indirection less, because it becomes the same as the `treasures` view for S_3 or S_4. In general, the transparency view can be updated to equal one of the successors, since that view was constructed using the same rules.

Unless the inlining is done cascading, the effect of this inlining is limited. Since performing this inline operation for each chunk operation to the origin site eventually results in a lot of traffic, this is not desirable. Instead, we envisage an inlining on demand, taking place during the querying process.

7.4.4 Improvement on the SQL implementation

The drawback of the previously introduced mapping of the Armada model to SQL, is that for each box, it needs a (physical) table in the database. This requires data to be physically grouped by the chunk function that describes it, and forces moves of tuples from and to the box tables upon chunking operations. In particular the copying from and to boxes on the same site is "expensive" because in principle the data doesn't physically "move" but needs to match the right box. In addition it introduces a space requirement (during the copy) which might be problematic for large boxes, or on machines scarce on resources.

A solution to the drawbacks sketched above, is to detach the actual storage from the box structure, and have the boxes being represented by a view instead. This allows to basically store the data in any table, or even multiple tables, as long as the box can be expressed by a view. It also gives the freedom to construct boxes from *existing* data by just defining views for them, instead of having to create them in an evolving matter. This feature in particular comes in handy when data from multiple sites is merged into one Armada instance.

The implementation of this solution is done using one table in the `armada` schema, named `data`. Since all data on a site is stored in that one table, there is now the need to create views that map back to the boxes again. Another table in the schema is introduced, the table `box`. This table simply holds the box IDs and their corresponding constraint as a string. This allows to use complex SQL functions, or simple conditions as checks in e.g. the WHERE-clause of a SELECT query.

7.5 Experimentation

Construction of Armada over relational systems stresses their capabilities to manage ever growing views during cloning, chunking, and combining. To assess this impact we conducted small experiments against MonetDB and PostgreSQL. We were unable to test MySQL as it does not support SQL standard syntax.

We can distinguish five different activities and related costs in the SQL approach taken by Armada.

table creation For each box, a table is created within the system. Because original tables get fragmented, extra tables need to be created. This may be an expensive operation. However, due to the expected frequency and other operations involved in chunking, of which table creation is a part, the impact of this cost is low.

data insertion To actually fill up the system, data is inserted in the boxes. Since this insertion itself is not any different from normal insertion, but that the site to insert to changes every once in a while, this cost is expected not to be any different from normal.

data moving On chunk operations, a part of the original box may be moved to another site. This involves a two-staged "copy and remove" action, with proper isolation and transportation of the data. This cost is obviously only involved if there is distribution by means of remote sites.

view creation Like table creation, view creation happens during the chunking process, and its costs get overshadowed by other operations of the chunking process. Hence, the impact of this cost is low.

view execution The generated views which cover the complete original tables become large. Query optimisers and planners may choke on such large views. Execution may become expensive due to the many tables (boxes) involved. However, since these views are the main route to query the system, the importance of the costs associated to them is high.

Query Complexity We conducted a number of experiments on our proposed SQL implementation of the Armada model. In particular the number of sites participating in Armada and the number of boxes that are created were taken as variables. The number of sites participating influences the necessary (total) communication. The number of boxes stresses the administration, and challenges its effectiveness.

As the number of sites that are populated by boxes increases, the need to contact other sites increases on average as well, as seen in Chapter 6. Such increase obviously comes with additional network costs. However, an increase of sites not necessary means an increase of communication. If queries target data that is located on one site, obviously not much communication is necessary. Moving targets, such as the last events (days) of a log file, typically are in a growing instance. However, ideally queries after such targets never need to consult more than one site.

With numerous boxes, the lineage of new boxes grows. Also, traversal paths may get very long in case a box has to be searched for. Limitations of the used SQL engines make it impossible to do any remote activity. Instead, we simulate everything local to the database, hence only being able to see query complexity and local execution. It also adds additional requirements as becomes clear later on.

Local Consistency When boxes get chunked, their contents is moved to other boxes. Hence, the original box remains empty and should be a pointer to the new boxes that were created. Conceptually, this is easy, as the chunked box can be replaced by an SQL view. However, databases that require their views to be correct, disallow tables that are used in a view to be dropped.

For this, first the view(s) that use the box should be removed. In Armada this yields in a problem. Ideally, the inactive boxes should just be replaced by a view, but the database forbids this action to take place. Of course, this only happens when the database has both the views and the boxes they depend on local, as only then it is easy to check. Unfortunately, this is always the case, because the view that describes the whole system, also describes the box that is chunked. However, if this view, the *worldview*, gets updated after the chunk operation, it should only list the offspring of the chunked box. So what remains pointing to the inactive box, is the worldview of the parent of the chunked box. Whenever this view is on the same site as the inactive box, the database can locally check and see that dropping the box breaks the dependency of that view. Hence, the only way out, is to recreate that view too.

Because this touches the view anyway, it is as expensive to immediately "inline" the new definition of the inactive box, instead of temporarily dropping it, replacing the box with a view and recreating it. When the parent *is* on a remote site, a view for the inactive box needs to be created, such that the view on the remote site still points to some object on the local site.

Table Replacement When, simulating an instance of Armada on a local site, local consistency checks influence how the operations take place. A DBMS checks the dependencies of objects that are in its catalog. Not only can it check for each foreign key constraint which table is necessary, but also for views it can determine which tables are used. An effect of this is that a table that is used in a view cannot be removed before that view is removed. The same holds for when a view is to be removed that another view uses.

With Armada, views get created to represent disappeared tables. When a box gets inactive, its corresponding table is replaced by a view. To do this, first the table is removed, then a view put in place. While the end situation in theory should be fine, the replacement of the table by a view is impossible from the DBMS's point of view. This is caused by Armada creating *worldviews* which are a view on the whole instance of Armada. As such, each table is part of one or more worldviews. Removing the table then breaks the dependencies of those worldviews. Replacement of the table by a view is not possible without the DBMS noticing in a local situation. If remote sites are referenced, the DBMS cannot see another site depends on an object, unless back references a maintained.

For the local situation, the worldviews that depend on the to be dropped object have to be dropped temporarily. Typically, for a given table, all worldviews that are on the site of the table itself, its sibling and all of its successors have to be dropped, as they point to the table being replaced. In addition to this, all views for the tables of the parent trail need to be dropped as they recursively depend on each other. Remember that a table is replaced with a view to its successors. When one of the successor tables is replaced with a view, it has the same problem as with the worldview. However, when the successor has been replaced by a view, there is a view to view dependency.

This is the case for every box in the parent trail of each table, since every box in the parent trail is inactive. This means that in order to be able to drop a view for an inactive box, its parent has to be dropped first, which recursively goes back to the root of the Armada instance. Hence, each operation in Armada, when simulated on a single site, is very expensive in terms of view destruction and creation.

Experiments An instance of Armada is a large collection of views and a few tables that hold the actual data. Not only are there many views, but also are the views very large in size, containing many levels of nested queries. In addition, many views depend on other views in the system. For these reasons, an instance of Armada is challenging in that it puts quite an unusual load on the DBMS.

We conducted a few experiments on a growing instance of Armada, to measure the effects of the views on the DBMS. Since we are not interested in the actual data load, but in the load of the metadata administration, we chose to use boxes of a small size containing between 5 and 15 tuples. With this size of the boxes, an instance of Armada grows relatively fast. In Table 7.1 the number of boxes created per number of tuples, and the inactive boxes is depicted in the first three columns. Inactive boxes in the SQL simulation are views, that due

tuples	boxes	inact.	M_t	M_1	M_2	M_3	P_t	P_1	P_2	P_3
250	37	18	3.50	3	3	3	7.21	1	1	1
500	71	35	13.68	5	5	5	23.29	2	2	2
750	107	53	39.80	8	8	8	67.15	6	3	3
1000	143	71	84.68	11	11	11	142.04	9	4	4
1250	179	89	178.07	15	15	15	245.24	13	4	5
1500	215	107	350.07	19	19	19	384.32	18	5	6
1750	255	127	614.58	24	24	24	590.16	24	6	7
2000	311	155	1131.23	31	31	31	956.27	31	8	9
2250	367	183	1852.58	40	40	40	1490.17	40	10	10
2250	simple		7.73	1	1	1	1.89	1	1	1

Table 7.1: Experimentation details.

to the aforementioned dependencies are recreated over and over again. As a result the 1000 tuples run contains 5184 view creation statements.

Origin Querying For each site, its worldview addresses every box in an instance of Armada. The difference in worldviews per site is in the amount of encoded lineage information. Sites with a larger parent trail, have more parent boxes encoded in their worldview. This is an advantage over sites with smaller parent trails, such as the root, as they have to "resolve" many boxes through views that point to views of their successors. For this reason, the origin is the most expensive cost-wise when querying its worldview. Hence, to get an idea of the maximum load introduced by the metadata administration of Armada, we measure query times on the origin site.

Querying a site is done with three different queries. First, a query that spans all (active) boxes is being run. It consists of a simple count of all tuples in the instance of Armada. Second, a range select query is performed. This theoretically allows the DBMS to skip a number of tables. Finally, a point query is performed. This always spans just one box in our experiment, since the key is required to be unique.

Table 7.1 lists the loading and query times for MonetDB/SQL 5.0.0_beta1 and PostgreSQL 8.0.8. The loading times are given in seconds in the columns M_t and P_t for MonetDB and PostgreSQL respectively. The columns M_1, M_2, M_3 and P_1, P_2, P_3 represent the average of 5 runs, preceded by 2 runs to assure the data is loaded and ready to be processed by the DBMS. The execution times are wall-clock times specified in hundreds of seconds.

Transaction Settings By accident the original loading set we created did not include transaction boundaries. Hence, the default transaction mode was used instead. In this mode after every statement a commit is issued. This was causing very long loading times for MonetDB. To solve this problem, the loading set was changed to include transaction boundaries, such that the whole load was seen as one transaction. This allowed us to compare the query results of PostgreSQL with MonetDB.

Apart from the loading times, which are dominated by view destruction and creation, the execution times on all of the sizes used in the experiment are far below a second per query. Yet they can grow up till 40 times the time needed to query a normal table in the case of a full scan in our largest example.

Even though the SQL statements to produce the instance of Armada do not contain any *check constraints* yet, PostgreSQL is able to avoid a full scan in case of range or point queries that only address one or a few of the tables.

Loading By observing the loading times, we can note that both PostgreSQL and MonetDB have difficulties with loading larger jobs, in two cases MonetDB actually crashes under the load. Jobs are generated by specifying a *sitesize* and a $tuplecount$. The former specifies how many tuples fit on a single site, where the latter specifies how many tuples are inserted in the instance of Armada. The jobs are a run with $sitesize = \{10, 50, 100, 500, 1000, 5000, 10000, 50000\}$ and $tuplecount = \{10, 50, 100, 500, 1000, 5000, 10000, 50000\}$. Not all combinations are generated, because of limitations in the generator. The avoided runs are those where the active box ratio (*tuplecount* divided by *sitesize*) is equal to or exceeds 1000. Note that the active box ratio is for many jobs the same. In such case the *sitesize* differs, which allows us to gain some insight on the effect of different data volumes.

While the loading times were not repeatedly measured, they are wobbly and must be read as a vague indication only. What we can conclude, however is that equal ratios take longer to load with more data, as one would expect.

Comparisons The graphs that depict the wall clock times for loading and the three queries, clearly show that for both MonetDB and PostgreSQL the loading times dominate over the query execution times by far. This doesn't come by a real surprise of course. Two peaks can be noted, for both databases. These peaks are at tests $10 - 5000$ and $100 - 50000$, which both have a ratio of 500. Obviously the amount of data involved makes a difference. For PostgreSQL this 10 times more data results in an almost 2 times longer loading time, while

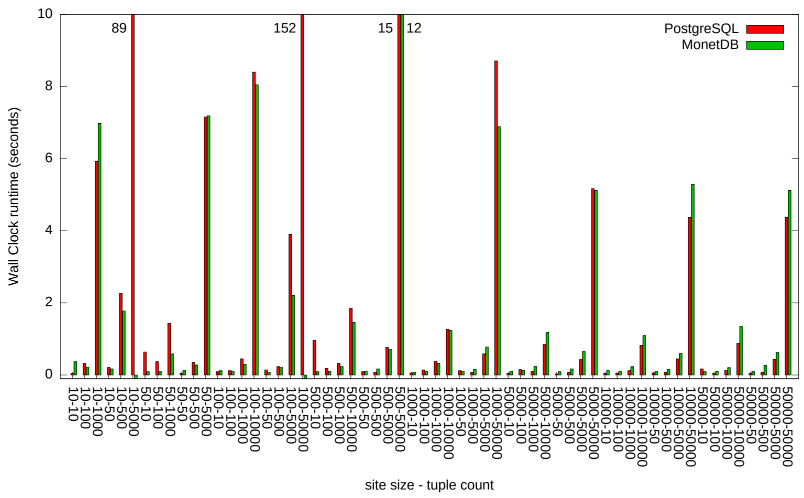


Figure 7.3: Load times.

MonetDB only needs about 10 seconds more, roughly $1/6^{th}$ of the loading time.

If we compare the loading times between the two DBMSs, as in graph 7.3, we can see that there is not a huge difference between the two. Apart from the two peaks discussed before, there is one extra peak for $500 - 50000$, ratio 100 that does not run within 10 seconds. The tendency here is that MonetDB is slightly faster, though not necessarily in all cases, such as e.g. $10 - 1000$. However, this might be very well be explained by startup times or similar, since the loading tests were not run multiple times in a row. Note that MonetDB crashes during loading for the $10 - 5000$ and $100 - 50000$ workloads.

Querying Query 1 consists of a count over all tuples in the relation over the system. This involves a scan over the full relation to count the tuples. The query and the results can be found in Figure 7.4. PostgreSQL clearly falls behind MonetDB in this phase, which is not surprising given the nature of both database systems. Note that there are no results for the workloads $10 - 5000$ and $100 - 50000$ since the loading fails.

The second query does the same count as the first query, but with an additional range predicate. The predicate is sufficiently small to only address a fraction of the Armada in most cases. The results from Figure 7.5 show a different

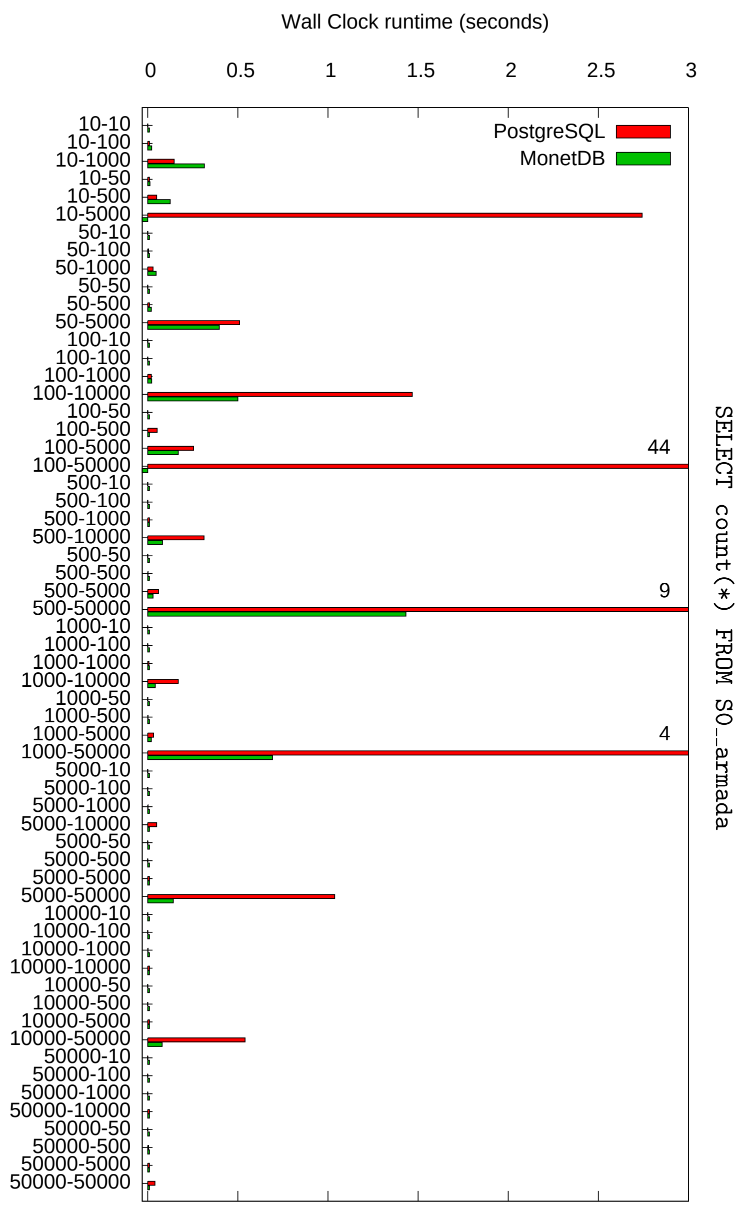


Figure 7.4: Query 1.

trend compared to the first query. Here PostgreSQL clearly outperforms MonetDB on almost every query. It looks like PostgreSQL is able to recognise only a limited number of tables is necessary to perform the query, whereas MonetDB seems to have equal performance as for query 1. This indicates it calculates the full Armada table, and performs the selection on that table to calculate the requested count.

The last query is a point query, targeting a single box. MonetDB not only falls behind PostgreSQL here, but also crashes on the $10 - 500$ and $100 - 5000$ workloads. Like for query 2, PostgreSQL is able to cut down the query time by identifying that the query only addresses a single table.

7.6 Limitations

Even though the SQL approach taken here deals with growth and querying, it is not a generic solution for a number of reasons. First and foremost does the SQL approach require a remote catalog/query mechanism to be available. As indicated before, not all databases have this functionality, and the many commercial databases have this available in some way, do not necessarily provide it in a form that is usable for Armada. PostgreSQL, MySQL, SQLite and MonetDB do not have said functionality to perform remote queries as of this writing. This observation makes a true heterogeneous implementation of Armada with the proposed SQL implementation impossible without a middleware layer to cater for the missing functionality.

A second issue is updating the data, like performing SQL INSERT, UPDATE and DELETE statements. The view constructions built by the SQL approach allow in principle for specific location of where updates should take place, in the same way as selects over the data can be located. However, updates of this kind over views is to the best of our knowledge not available. To a very limited extent, updates on views are possible in most commercial database systems. The typical functionality here is to allow updates on views that do projections and/or selection only and updates that only involve one of the tables used in the view. Since the Armada SQL model does UNIONs of tables, execution of updates on those views require some (semantic) knowledge behind the views and their purpose in Armada. Also this could be solved using a middleware approach, but it would require a full SQL interpreter in the middleware software, including the Armada logic. Such approach would not require any generic (SQL) approach, since the middleware translates all functionality to the target database. This was not the focus of the proposed SQL approach.

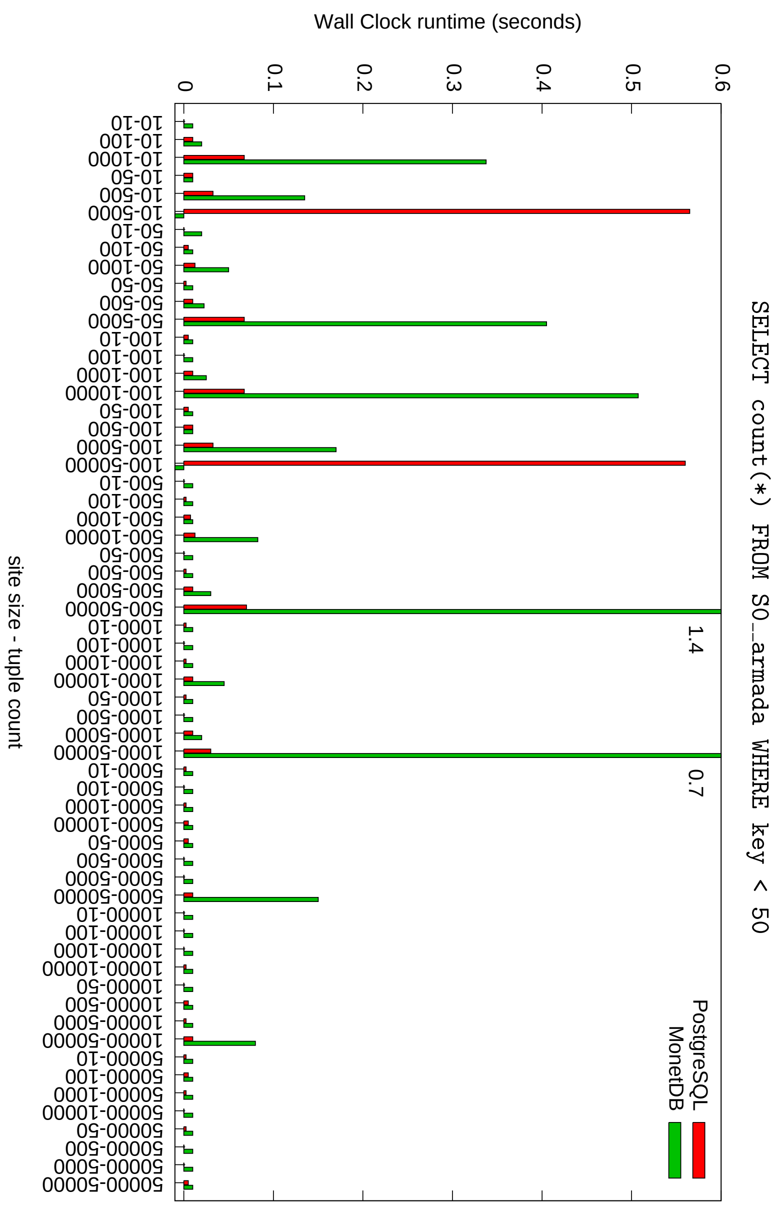


Figure 7.5: Query 2.

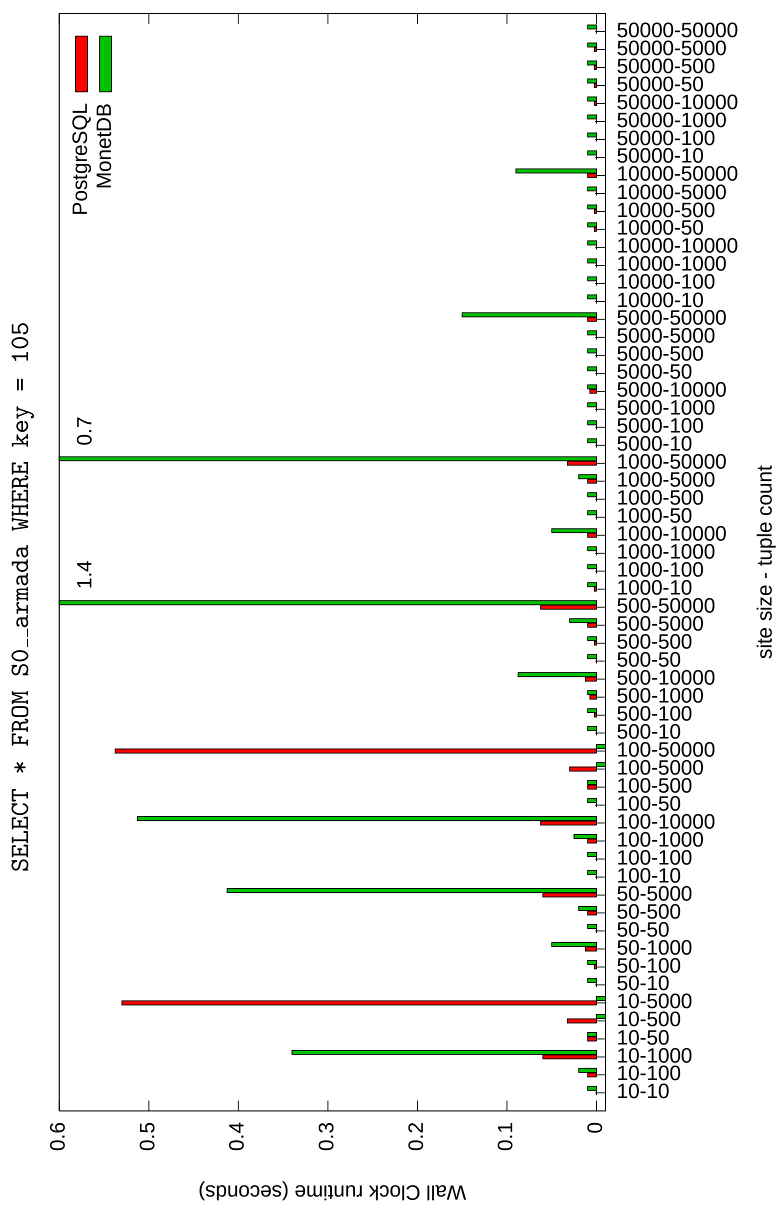


Figure 7.6: Query 3.

workload	M_l	M_1	M_2	M_3	P_l	P_1	P_2	P_3
10-10	0.37	0.01	0.01	0.01	0.05	0	0	0
10-100	0.22	0.0225	0.02	0.02	0.32	0.01	0.01	0.01
10-1000	6.98	0.315	0.3375	0.34	5.93	0.1475	0.0675	0.06
10-50	0.17	0.0125	0.01	0.01	0.21	0.01	0.01	0.01
10-500	1.78	0.125	0.135	crash	2.27	0.05	0.0325	0.0325
10-5000	crash	crash	crash	crash	89.27	2.7425	0.565	0.53
50-10	0.09	0.01	0.02	0.01	0.64	0	0	0
50-100	0.10	0.01	0.01	0.01	0.37	0.005	0.005	0.0025
50-1000	0.59	0.0475	0.05	0.05	1.44	0.03	0.0125	0.0125
50-50	0.13	0.01	0.01	0.01	0.04	0	0.0025	0
50-500	0.28	0.02	0.0225	0.02	0.35	0.01	0.01	0.01
50-5000	7.19	0.3975	0.405	0.4125	7.16	0.51	0.0675	0.06
100-10	0.12	0.01	0.01	0.01	0.09	0	0.005	0
100-100	0.10	0.01	0.01	0.01	0.12	0	0	0
100-1000	0.30	0.0225	0.025	0.025	0.45	0.02	0.01	0.01
100-10000	8.05	0.5	0.5075	0.5125	8.40	1.4675	0.0675	0.0625
100-50	0.08	0.01	0.01	0.01	0.14	0	0.005	0
100-500	0.22	0.01	0.01	0.01	0.23	0.0525	0.01	0.01
100-5000	2.21	0.17	0.17	crash	3.90	0.255	0.0325	0.03
100-50000	crash	crash	crash	crash	152.05	44.18	0.56	0.5375
500-10	0.09	0.01	0.01	0.01	0.97	0	0	0
500-100	0.10	0.01	0.01	0.01	0.19	0	0.0025	0.0025
500-1000	0.23	0.01	0.01	0.01	0.32	0.01	0.0075	0.0075
500-10000	1.46	0.0825	0.0825	0.0875	1.86	0.3125	0.0125	0.0125
500-50	0.11	0.01	0.01	0.01	0.09	0	0	0
500-500	0.17	0.01	0.01	0.01	0.08	0.0025	0.0025	0.0025
500-5000	0.72	0.03	0.03	0.03	0.77	0.06	0.01	0.01
500-50000	12.04	1.4325	1.425	1.4225	15.39	9.23	0.07	0.0625
1000-10	0.08	0.01	0.01	0.01	0.06	0	0.0025	0.0025
1000-100	0.10	0.01	0.01	0.01	0.14	0	0	0
1000-1000	0.32	0.01	0.01	0.01	0.38	0.0075	0.0025	0
1000-10000	1.24	0.0425	0.045	0.05	1.27	0.17	0.01	0.01
1000-50	0.11	0.01	0.01	0.01	0.12	0	0.0025	0
1000-500	0.16	0.01	0.01	0.01	0.07	0.0025	0	0
1000-5000	0.78	0.02	0.02	0.02	0.59	0.0325	0.01	0.01
1000-50000	6.89	0.6925	0.69	0.6975	8.71	4.4775	0.03	0.0325
5000-10	0.11	0.01	0.01	0.01	0.04	0	0.0025	0
5000-100	0.13	0.01	0.01	0.01	0.15	0	0	0
5000-1000	0.24	0.01	0.01	0.01	0.10	0.0025	0.0025	0
5000-10000	1.18	0.01	0.01	0.01	0.86	0.05	0.005	0.0075
5000-50	0.09	0.01	0.01	0.01	0.03	0	0.005	0
5000-500	0.17	0.01	0.01	0.01	0.07	0	0	0
5000-5000	0.65	0.01	0.01	0.01	0.43	0.01	0	0
5000-50000	5.12	0.1425	0.15	0.15	5.17	1.0375	0.01	0.01
10000-10	0.13	0.01	0.01	0.01	0.04	0	0	0
10000-100	0.11	0.01	0.01	0.01	0.05	0	0	0
10000-1000	0.23	0.01	0.01	0.01	0.12	0.0025	0	0
10000-10000	1.09	0.01	0.01	0.01	0.82	0.01	0.0025	0
10000-50	0.10	0.01	0.01	0.01	0.06	0	0	0.0025
10000-500	0.16	0.01	0.01	0.01	0.07	0	0.005	0.0025
10000-5000	0.60	0.01	0.01	0.01	0.45	0.01	0.0025	0
10000-50000	5.29	0.08	0.08	0.09	4.37	0.54	0.01	0.01
50000-10	0.09	0.01	0.01	0.01	0.17	0	0.0025	0
50000-100	0.10	0.01	0.01	0.01	0.04	0	0.005	0
50000-1000	0.20	0.01	0.01	0.01	0.13	0	0.0025	0
50000-10000	1.34	0.01	0.01	0.01	0.87	0.01	0	0.0025
50000-50	0.10	0.01	0.01	0.01	0.04	0	0.0025	0.0025
50000-500	0.27	0.01	0.01	0.01	0.07	0.005	0	0.0025
50000-5000	0.62	0.01	0.01	0.01	0.44	0.01	0	0.0025
50000-50000	5.12	0.01	0.01	0.01	4.37	0.04	0.005	0

Table 7.2: Extended experiment times.

Lastly, for operations to be executed somewhat efficiently, and to have autonomy in the sense that the databases are not depending on each other, each database must support to have views with “unknown” non-local objects that can only be checked by contacting a remote site. That check, however should not be performed for each and every operation, but instead left to the agent in the system to discover.

A final note on the viability of the SQL approach deals with optimisations specific to Armada, in particular originating from the knowledge stored. For queries that just address a particular range that can be matched upon certain boxes, it is evident that querying all boxes in the Armada is a waste of resources. To avoid this waste, execution should skip boxes that cannot contain requested data. For this, the chunk functions need to be used at the execution to skip contacting remote sites were this is deemed impossible.

Summary

Using the SQL language, we aimed for a straight-forward and relatively non-intrusive implementation of the Armada model in existing database management systems. In particular SQL *views* are a central concept within the chosen implementation approach. We demonstrated how the *chunk*, *clone* and *combine* operations of the Armada model can be mapped onto SQL, and which actions are required to perform the respective operations.

The distribution aspect of Armada is hidden under the assumption that there is a notion of “remote tables”. This notion allows a database to retrieve data from a table which is on a remote site as if it were local to the database. This functionality is only available in a limited set of traditional databases to date, to the best of our knowledge. Due to the nature of our implementation, Armada can be easily simulated on one machine where all boxes are local first. Whenever remote tables become available, the Armada implementation can easily switch to using real distribution of the data.

The two database systems that we have experimented with, have shown that the approach taken puts an unusual load on the systems, sometimes even resulting in crashes. While the construction of an Armada takes a lot of time, this can be accepted in general, since it is not a frequently occurring operation. The query performance instead is important. Compared to a single large table the fragmented Armada is slower, especially when all boxes in the Armada need to be consulted. Given that it is more probably that this is not the case, the results are acceptable in that area. SQL engines can be optimised for a more efficient use of large SQL views. Crashes under the load produced by the queries and views are of course a bad sign that the underlying engine is not able to handle the size of the queries, which needs to be tackled first.

Loading succeeds using some tricks, and querying works reasonably well. However, inserting and updating data does not work due to limitations on the SQL views observed on many SQL implementations. The loading tricks also affect the matter in which a database can freely evolve, as in extend itself to others in the cluster. Additionally, the SQL implementation does not allow for the active client model, but turns each server into a recursive query resolver itself. This obviously breaks the autonomy as defined by the Armada model.

8 MonetDB MAL Approach

8.1 Introduction

In previous chapters, the Armada model and a mapping of the model to SQL statements have been described. As shown there, SQL itself is not powerful enough to allow an implementation of Armada without having to make modifications to existing systems. Instead of focussing on the SQL layer of a database system, in this chapter we explore an implementation of the Armada model on a deeper level towards the core. For this exploration we use MonetDB and its kernel language *MAL*. Instead of implementing Armada on a user visible level, such as in SQL tables, in this approach we make numerous Armadas on an abstract level, hidden from the user. Being on a deeper level, allows for a better grip on the actions taken in the Armada model, with possible better performance as result. As entry point we assume the use of SQL, but tables no longer are Armadas. Instead the underlying representation of the database uses the Armada model to distribute the components it uses.

Typically, each SQL query is translated into MAL statements within MonetDB for execution. The units that MonetDB works with, called BATs, are target for distribution. Each SQL table consists of multiple of such BATs. Figure 8.1 depicts the architecture that we use in this chapter. Clearly, a user interacts with the database using plain SQL. Internally, this is converted to MAL code, and the Armada model is implemented via MAL programs that reference remote sites. The actual work is performed by the site that the user connected to. It hence can be considered to be the agent of the system.

SQL catalog In the modular architecture of MonetDB, the SQL compiler references BATs via the catalog. To make the SQL compiler unaware of any Armada activities, catalog entries can be adjusted such that they point to BATs which represent Armadas.

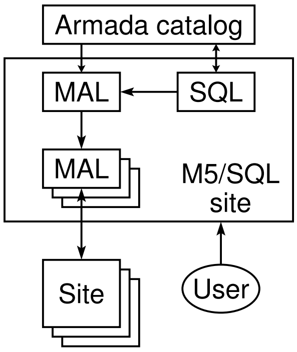


Figure 8.1: Ideal architecture of a MAL-based Armada implementation.

This approach keeps the SQL compiler unmodified, and hence is a cheap solution, focussing on the MAL layer. However, this approach limits the possibilities to communicate with the user about decisions that can be made in the Armada execution. For user feedback, partial execution is necessary and some support to have a user decide on how to continue execution. The SQL standard does not have any means for achieving this, and hence no handles for this are available. Instead, we focus on full execution without decisions such that we are compatible with the (unmodified) SQL compiler.

8.2 MonetDB

The MonetDB database system has been an open source product since 2000, hosted on SourceForge.net. Even though it is open sourced, a clear research focus is the drive behind the database. This becomes apparent when considering its very non-traditional design, aimed at non-traditional workloads. Core of MonetDB is its main-memory processing of data. As this puts restrictions on when data can be processed efficiently, a full vertically fragmented storage model is in place that allows to only process certain conventional columns, instead of full rows. The quest for performance on large workloads with MonetDB has resulted in sophisticated in-memory algorithms that are CPU-tuned, with an accompanying architecture to ensure the CPU needs not to idle, wasting precious time.

The vertical fragmentation of MonetDB makes it a member of the class of column-oriented data-stores among [64, 47]. In MonetDB, every relational table is represented as a collection of *Binary Association Tables* (*BATs*). For a relation R of k attributes, there exist k BATs, each BAT storing the respective column as a collection of `key-attr` pairs. The system-generated `key` identifies the relational tuple that attribute value `attr` belongs to, i.e. all attribute values of a single tuple are assigned the same `key`. Typically, key values form a dense ascending sequence representing the *position* of an attribute value in the column. This enables MonetDB to use fast positional lookups in a BAT given a key (such as for tuple reconstruction) and to avoid materialising the key part of a BAT in many situations completely. BATs are stored as dense tuple sequences, to enable fast in-memory processing.

The database kernel consists of a large collection of highly tuned algorithms to evaluate basic binary relational operators. All operators work independently and produce a fully materialized result. Each operator includes a runtime optimiser to exploit storage properties of operands, like sortedness, uniqueness and data type. Heavy code expansion is used to further reduce the overhead of interpretation.

MAL The primary textual interface to the MonetDB kernel is a simple, assembler-like language, called MAL. The language reflects the virtual machine architecture around the kernel libraries and has been designed for speed of parsing, ease of analysis and ease of target compilation by query compilers. The language is not meant as a primary programming or scripting language, such use is even discouraged.

Furthermore, a MAL program is considered a specification of intended computation and data flow behaviour. It should be understood that its actual evaluation depends on the execution paradigm chosen in the scenario. The program blocks can both be interpreted as ordered sequences of assembler instructions or as a representation of a data-flow graph that should be resolved in a data flow driven manner. The language syntax uses a functional style definition of actions and mark those that affect the flow explicitly. Flow of control keywords identify a point to change the interpretation and denote a synchronisation point.

8.2.1 MonetDB/SQL

A user typically uses the SQL language to interact with databases. This language is in general easier to understand for humans than programming or scripting languages are. However, being a language geared towards humans, it is not sufficient for execution by a machine. Hence, the SQL query is compiled into the lower level MAL language, that in turn does a step deeper downwards to machine language.

An SQL query is compiled into MAL statements. For this compilation, logical table and column properties need to be known, such that can be checked whether a query is using existing objects, and whether they are of the right type for what they are used for. This information is typically contained in the catalog of the SQL database. In MonetDB/SQL the catalog is implemented by a few *system tables* that in principle are ordinary SQL tables containing information about all objects present in the database. The vertical fragmented nature of MonetDB has the effect of having at most single column tables. The MonetDB/SQL implementation hides this limitation, by using a BAT for each column of an SQL table, and keeping the administration of the set of BATs that make up a single SQL table.

The output produced by an SQL compiler consists of sizeable MAL programs, mostly comprising binary relational algebra operations. They have already been optimised using the basic relational rewrite rules, such as selection push-downs by the SQL compiler. The MAL program also takes care of glueing together the relational containers with pending inserts, deletes and updates to represent the latest consistent snapshot. Furthermore, the MAL program is decorated with all information needed to optimise and execute the query without access to the SQL catalog. This information takes the form of function calls with all-constant arguments and properties linked with variables.

Figure 8.2 illustrates the MAL plan produced by the compiler for the query `SELECT count(*) FROM R, S WHERE R.key = S.key AND R.Key < 23`. The query is translated into a cached function, which is called with argument 23. The body of the function is a linear representation of the logical expression. The first section locates the BATs that represent the two tables R and S. It also obtains the reference to the pending updates and deletes, which are consolidated in the algebraic section. The major part is the binary relational algebra plan.

The effective result of the SQL compilation phase is a MAL plan that references columns that are necessary to compute the answer to the original SQL query. The Armada implementation at the MAL level targets individual BATs. Since the compiled SQL queries address BATs, this is where the Armada comes in. Obviously, as each Armada in this case is a single column, vertical fragmentation is not possible in this scheme.

```
function user.s2_0(A0:sht):void;
  _2:bat[:oid,:int]{rows=1:lng,notnil=true} := sql.bind("sys","r","key",0);
  _7:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","r","key",1);
  _10:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","r","key",2);
  _14:bat[:oid,:oid]{rows=0:lng} := sql.bind_dbat("sys","r",1);
  _29:bat[:oid,:int]{rows=1:lng,notnil=true} := sql.bind("sys","s","key",0);
  _31:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","s","key",1);
  _33:bat[:oid,:int]{rows=0:lng,notnil=true} := sql.bind("sys","s","key",2);
  _36:bat[:oid,:oid]{rows=0:lng} := sql.bind_dbat("sys","s",1);
  _9 := algebra.kunion(_2,_7);
  _12 := algebra.kdifference(_9,_10);
  _13 := algebra.kunion(_12,_10);
  _15 := bat.reverse(_14);
  _16 := algebra.kdifference(_13,_15);
  _17 := A0;
  _18 := calc.int(_17);
  _19 := algebra.uselect(_2,nil:int,_18,false,false);
  _22 := algebra.uselect(_7,nil:int,_18,false,false);
  _23 := algebra.kunion(_19,_22);
  _24 := algebra.kdifference(_23,_10);
  _25 := algebra.uselect(_10,nil:int,_18,false,false);
  _26 := algebra.kunion(_24,_25);
  _27 := algebra.kdifference(_26,_15);
  _28 := algebra.semijoin(_16,_27);
  _32 := algebra.kunion(_29,_31);
  _34 := algebra.kdifference(_32,_33);
  _35 := algebra.kunion(_34,_33);
  _37 := bat.reverse(_36);
  _38 := algebra.kdifference(_35,_37);
  _39 := bat.reverse(_38);
  _40 := algebra.join(_28,_39);
  _41 := calc.oid(0@0);
  _43 := algebra.markT(_40,_41);
  _44 := bat.reverse(_43);
  _45 := aggr.count(_44);
  sql.exportValue(1,"sys.","count_","int",32,0,6,_45,"");
end s2_0;
```

Figure 8.2: `SELECT count(*) FROM R, S WHERE R.key = S.key AND R.key < 23.`

```
function armada.box0(){inline,active}:bat[:oid,:int];
    armada.ensureActive("myhostname","armada.box0");
    b0{remote,armada} := remote.get("myhostname","internal_bat_24");
    armada.chunk(b0{remote,armada},0,-1);
    return b0{remote,armada};
end box0;
```

Figure 8.3: First generation wrapper function.

8.3 Static MAL Armada implementation

MAL code produced by the SQL compiler follows in general a sequence of *bind* calls, BAT operations and a final result for the user. The bind calls refer to local BATs for the SQL compiler that are maintained and kept aligned where necessary. Since these bind calls are just visible in the MAL code, code transformers can detect and replace these bind calls with Armada specific calls to activate special use of BATs as part of an Armada. Since this is done by code transformers, the SQL compiler is unaware of this change. This is the starting point for an Armada at the level of MAL code.

To have Armada working at the MAL level, we need to encode the Armada model in MAL structures. The MAL language has the notion of function routines that have one or multiple return values after a call. We can use functions to represent a box. Normal BATs are not sufficient to represent a box, since in the Armada model boxes can become inactive, which means such boxes are redirects to other boxes. BATs cannot have this behaviour, but functions, on the other hand, can be changed to reflect this state. Assuming that we have an *armada* name space, box functions with an unique name can be placed in this name space. An Armada "BAT" is addressable via its box name only. This means that a BAT is wrapped by a function stored in the *armada* name space.

Wrapper Function In Figure 8.3 such function is shown. It contains extra information that in later stages can be used to steer the process of query execution. Starting with the declaration of the function, we see two properties, *inline* and *active*. The first property is a hint that the contents of the function can (and should) be easily inlined in other code which calls the function. Inlining allows code transformers and optimisers to easily consider the function as a part of the original code and do changes that go beyond the scope of the inlined function.

The second property, *active*, represents the state of the box, whether it is containing data (active) or whether it is just a redirect to other boxes (inactive) after an operation has been applied. Recall from the Armada model that a box can only participate in an operation once. Hence, when the box is marked as inactive, its contents (redirects) never changes.

Assertions The first statement in the function is an assertion statement, that ensures the function is representing an active box. While this may seem superfluous, its use becomes apparent in the light of function caching. A function that is once read by a remote site may be cached or implicitly cached due to the inlining property. A box can become inactive, hence so can the function. This happens when the function is for instance chunked. Caching a function which represents an active box leads to problems once the box becomes inactive. A site which uses an outdated (cached) version of the function does not notice that the used BAT is no longer the full dataset, but only a part instead. Obviously this is very much undesirable, hence it needs to be ensured that the cached function is legally used in case of an active box. `armada.ensureActive` is a no-operation function for the program in the function. When it fails to ensure that the given function is active, it forces a reconsideration of the query (MAL) plan, starting from the last known state where the function which failed to ensure being active was introduced. We discuss the details of this *trap* in the optimiser framework further on in Section 8.3.1.

Remote Retrieval The next line in the function gets a BAT from the remote site and assigns its copy to `b0`. The `get` effectively copies the data from the remote BAT to the local site and makes it available as `b0`. The copying can be delayed to the first moment that `b0` is used. Further optimisations regarding fetching only parts (to avoid doing the entire BAT) are of later concern, at the stage where we know upfront that we only need a part of the BAT. For now, it keeps the functionality of the `get` function limited to making the remote BAT available. Note that the `get` call is constructed in such a way that also when the contents of the function is inlined on another site, the statement still works and results in the same data. The mechanism should be clever enough to simply issue a local bind in case the “remote” host is the same as the local host.

Function Encoding To encode the functions that are applied to the boxes, as defined in the Armada model, special meta-statements are necessary. Since a box does not contain any more data than their function describes, additional selections over the boxes that match the function are a waste of efforts. The adding such *select* statement in the function technically allows for optimisers to detect the selection boundaries and take them into account, such ordinary select statements are hard to distinguish from those statements that are really necessary to perform. To avoid this potential performance pitfall, we use meta-statements for this that *describe* what data can possibly be in a given BAT. The `armada.chunk` statement does this for BAT `b0` in the above MAL function. It describes a range chunk function and specifies what (numeric) range is being used.

Returning Finally, `b0` is being returned to the caller. Note that `b0` carries the properties *remote* and *armada*. The first marks the BAT as being a copy of a remote BAT somewhere. The second marks the BAT being a result of and owned by Armada. Both properties can be used later in the process to treat the BAT properly on optimisations and operations.

8.3.1 Use of Optimiser Plan Stacks

The MonetDB distribution comes with a large collection of optimiser modules. They are developed up to the point that they could be used to experiment with the optimiser software infrastructure. They are highly targeted at a particular problem. Table 8.1 shows the modules forming the optimiser pipeline for SQL plans.

The MAL language does not imply a specific optimiser to be used. Instead, calls to specific optimiser routines is part of the MAL program produced by the front-end compiler. They are, however, evaluated during the optimiser phase only.

Plan Stacks Optimisers have the freedom to change the code, provided it is known that the plan derived is invariant to changes in the environment. In particular, an optimiser may leave behind calls to other optimiser routines. When all optimiser calls have been dealt with, the query plan is cached and ready for execution. The alternative plans are collected as a stack of MAL program blocks. The *plan stack* can be inspected for a posterior analysis of optimiser behaviour. Alternatively, the stack may be pruned and re-optimised when appropriate from changes in the environment. An example of such change is whether a BAT is empty or not. Big parts of the code may be disabled and removed if an input BAT is empty, however, if the BAT becomes non empty later on when reusing the

inline	inlines functions identified as such
remap	locates hardwired multiplex operations
costModel	inspects the SQL catalog for size information
coercions	performs static coercions
emptySet	removes all empty set expressions
access modes	ensures that BATs for update are writeable
aliases	removes alias assignments
common terms	searches for common terms and retains one only
accumulators	re-uses BATs to hold the result of an expression
joinPath	searches multiple joins and glues them together
deadcode	removes all code not leading to used results
reduce	reduces the stack space for faster calls
garbageCollector	injects calls to free up space
multiplex	translates multiplex operations to iterators

Table 8.1: The MonetDB/SQL optimiser pipeline.

same (optimised) plan, a re-evaluation of the original plan is necessary, since it obviously is not valid any more. Re-evaluation may, however, not always have to be done completely from scratch. It may only be necessary to re-evaluate from a given point in the stack. This is typically the case for the `armada.ensureActive` function. In case the check fails, a re-evaluation has to be made starting from the point where the statement was introduced. For the Armada case, this is the point where the remote function was inlined in the query plan.

Labels When the `armada.ensureActive` statement is inlined, its invocation is changed to refer to the plan right before the inline operation. This allows the operation to jump back to the right plan in the stack to start re-evaluating. For this to work, each plan in the stack is labelled. MAL properties control whether a reference to a label should be made or whether the statement has to be kept as is. The transformer that sets the label for the `ensureActive` statement looks for the properties `func` and `label`. If one or both are missing or the value of `func` is not equal to the current function name, the statement is considered to be new. In this case the `func` property is set to the name of the current MAL function. The `label` property is set to the label of the current plan in the stack, that is the label of the last known plan. This is not the plan that is currently being generated by the transformer. Using this scheme the jump point for the `ensureActive` statement is being set once it is introduced or inlined into

```
sql oranges => columns "id" is t_id, "name" is t_name, "value" is t_value
bat t_id    => "december" on "amalia"
bat t_name  => "june" on "alexia"
bat t_value => "april" on "ariane"
```

Figure 8.4: The catalog state.

another plan. This results in the desired behaviour where a jump is being made to the right label and an optimiser is able to simply determine whether the jump label has to be set or not.

8.3.2 Analysis Use Case

Using the previously described techniques to manipulate functions on the MAL level, we can build an Armada system as follows. For this example we assume that the catalog contains information about an SQL table “oranges”, having three columns, “id”, “name” and “value”. Each of the three columns are mapped to a named Armada BAT, “december” on host “amalia”, “june” on host “alexia” and “april” on host “ariane” respectively. Figure 8.4 schematically represents this catalog.

The state of the catalog specifies that there exist three sites that have an Armada BAT. Those BATs are initially set up on their sites, and made accessible by a wrapper function. The wrapper functions are like Figure 8.3, but for “december” on “amalia”, a chunk operation was applied. The three transitions the function “december” made are depicted in Figure 8.5.

In our example we only consider the last state of the “december” function to be used. In other words our example runs only after “december” has been chunked. As can be seen in Figure 8.5, “december” has become an inactive box, marked by the absence of the “active” property. Any following changes do not affect the box being active or not, following the Armada model. The definition of the newly used boxes, such as “january” on host “maxima” has been inlined in the “december” function, including the guarding assertion to make sure no out of date definition of the function is used. The `armada.chunk` operations indicate how the chunk was performed using a range function. The guards may cause the function to be re-evaluated starting from the second function. As can be seen, the `armada.ensureActive` call for the “december” function itself is dropped in the second function, as it is no longer necessary to ensure that the box is active. The box is inactive, which means its actual contents is not going

```
function amalia.december{inline,remote,active}():bat[:oid,:int];
    armada.ensureActive("amalia.december");
    b1 := remote.bind("amalia","b7");
    ret := armada.chunk(b1,0,-1);
    return ret;
end december;

function amalia.december{inline,remote}():bat[:oid,:int];
    b1 := amalia.regina();
    b2 := maxima.january();
    return ret := algebra.sunion(b1,b2);
end december;
    armada.resolve("amalia.regina");
    armada.resolve("maxima.january");

function amalia.december{inline,remote}():bat[:oid,:int];
    armada.ensureActive("amalia.regina");
    b1 := remote.bind("amalia","b7");
    b3 := armada.chunk(b1,0,20);
    armada.ensureActive("maxima.january");
    b2 := remote.bind("maxima","b32");
    b4 := armada.chunk(b2,20,-1);
    return ret := algebra.sunion(b3,b4);
end december;
```

Figure 8.5: Transitions for the function “december” on host “amalia”.

```
function user.s1_0():void;
    _8:bat[:oid,:int] := armada.bind("t_id");
    _15:bat[:oid,:str] := armada.bind("t_name");
    _18:bat[:oid,:int] := armada.bind("t_value");
    _11 := algebra.markT(_8,0@0);
    _12 := bat.reverse(_11);
    _13 := algebra.join(_12,_8);
    _16 := algebra.join(_12,_15);
    _19 := algebra.join(_12,_18);
    _20 := sql.resultSet(3,1,_13);
    sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
    sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
    sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
    sql.exportResult(_20,"");
end s1_0;
    armada.resolve("user.s1_0");
```

Figure 8.6: Simplified initial SQL query plan for `SELECT * FROM oranges`.

to become wrong, at most out of date. Hence, the function can be freely copied and cached. Of course the newly added guards in the third function make the function body itself potentially incorrect again. This is due to the contents of the referenced functions being inlined.

Query Execution In the described setting, an agent called “trix” executes the SQL query `SELECT * FROM oranges`. This query translates to MAL code as in Figure 8.6. The figure shows simplified code, leaving out the delta administration for inserts and deletes. Also, some optimisers have been run. Normally also the empty set optimiser runs, but in this example it has been disabled, for it would remove empty results, thereby making it harder to see what the code is doing. The code in the figure essentially just binds to the columns and prepares them to be aligned before being added to a final result where additional metadata is stored. At the bottom of the code an `armada.resolve` call is found. It processes the `armada.bind` calls that are inserted as a replacement of `sql.bind` calls. This processing either results in the call being replaced by a simple `sql.bind` call for non-Armada BATs, or the (remote) function that maps the requested Armada BAT.

Figure 8.7 depicts the situation after resolving the `armada.bind` calls. In

```
function user.s1_0():void;
    _8:bat[:oid,:int] := amalia.december();
    _15:bat[:oid,:str] := alexia.june();
    _18:bat[:oid,:int] := ariane.april();
    _11 := algebra.markT(_8,0@0);
    _12 := bat.reverse(_11);
    _13 := algebra.join(_12,_8);
    _16 := algebra.join(_12,_15);
    _19 := algebra.join(_12,_18);
    _20 := sql.resultSet(3,1,_13);
    sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
    sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
    sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
    sql.exportResult(_20,"");
end s1_0;
    optimizer.inliner("user.s1_0");
```

Figure 8.7: Resolved query from Figure 8.6.

our case, all binds are replaced with calls to our previously defined Armada BAT functions. Not surprisingly, these calls can be inlined to further expand the plan, like we did in Figure 8.5. Eventually, after all code has been expanded, the final plan becomes as in Figure 8.8. In that plan, in total three `armada.ensureActive` statements are present. These statements are potentially invalidating the plan. More importantly, they need to be checked during execution, which requires (network) communication with the involved site. This obviously is an inevitable pity, since it adds extra network costs. When the plan is being constructed, the site is contacted for the first time. Then before the data is being requested, it is checked whether the plan is up-to-date by checking with the site. Finally, the site is contacted for the last time to retrieve the data. In case any up-to-date check fails, even more communication costs are involved. Obviously, it is desirable to reduce the number of communications with sites.

8.4 Stepwise Dynamic Inlining

While the before described approach is clear for analysis of queries to be performed on an Armada, it imposes a different agent strategy than the Armada model proposes. In terms of agent communications with remote servers the Ar-

```
function user.s1_0():void;
    armada.ensureActive("amalia.regina");
    _1 := remote.bind("amalia","b7");
    _2 := armada.chunk(_1,0,20);
    armada.ensureActive("maxima.january");
    _3 := remote.bind("maxima","b32");
    _4 := armada.chunk(_3,20,-1);
    _8:bat[:oid,:int] := algebra.sunion(_2,_4);
    armada.ensureActive("alexia.june");
    _14 := remote.bind("alexia","b26");
    _15:bat[:oid,:str] := armada.chunk(_14,0,nil:str);
    armada.ensureActive("ariane.april");
    _17 := remote.bind("ariane","b10");
    _18:bat[:oid,:int] := armada.chunk(_17,0,-1);
    _11 := algebra.markT(_8,0@0);
    _12 := bat.reverse(_11);
    _13 := algebra.join(_12,_8);
    _16 := algebra.join(_12,_15);
    _19 := algebra.join(_12,_18);
    _20 := sql.resultSet(3,1,_13);
    sql.rsColumn(_20,"armada.oranges","id","int",32,0,_13);
    sql.rsColumn(_20,"armada.oranges","name","varchar",24,0,_16);
    sql.rsColumn(_20,"armada.oranges","value","int",32,0,_19);
    sql.exportResult(_20,"");
end s1_0;
```

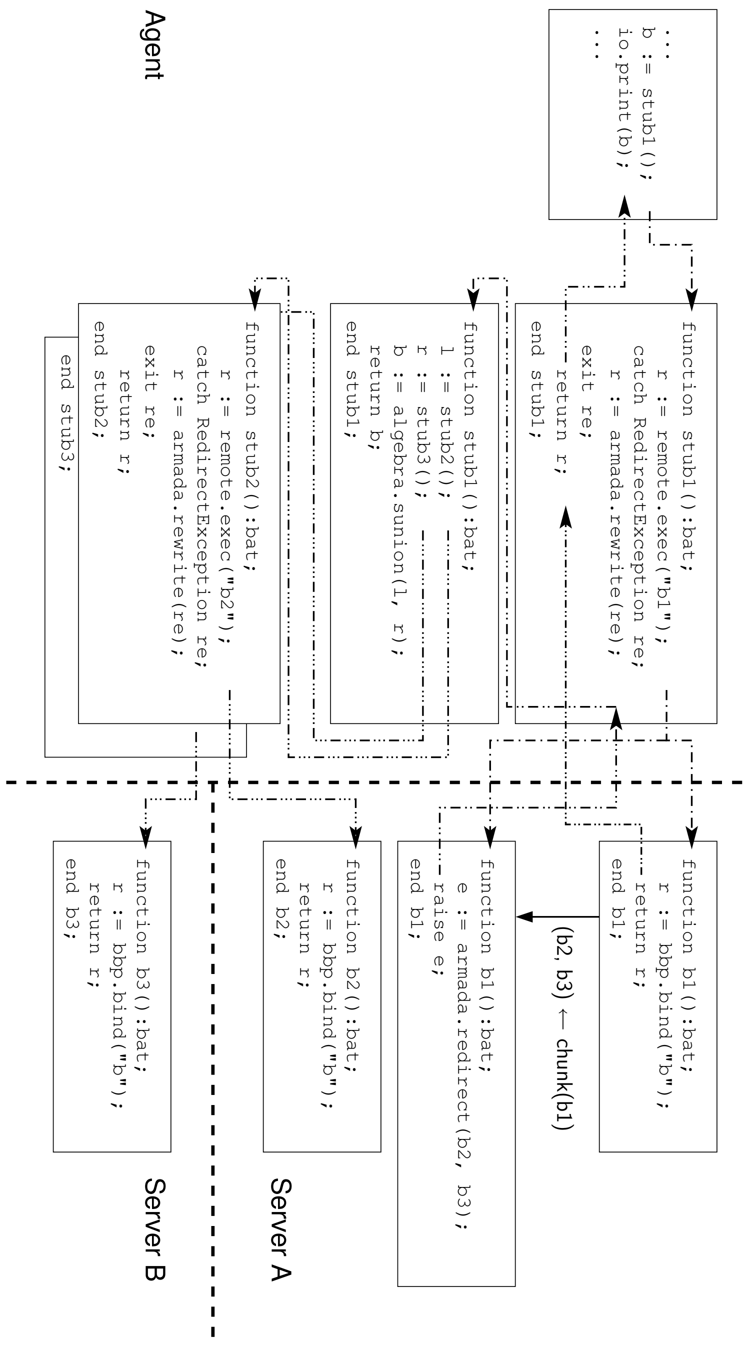
Figure 8.8: Final expanded query plan.

mada agent is more efficient, and also efficiency through caching is significantly easier. The key of an Armada agent is that it detects an outdated box when it asks it for its data. Not only does this mean no total plan is generated before execution, but also that an agent has no notion of whether a box is active or inactive. When an agent requests a box for its data, it simply receives this data, or a redirection where to find this data. The agent simply refines its plan at runtime with this received redirect. Based on this property, another approach was devised that inhibits this plan refinement at runtime behaviour.

Dynamic environments require dynamic execution models to be efficient. *Stepwise dynamic inlining* is a query execution strategy where the query plan is incrementally built during query execution. It can be seen as a refinement of the query plan that is made during execution. The plan is seen as just an approximation or as out of date.

Incremental Optimisations The architecture imposed by the Armada model requires an agent to be an active player during query execution. Not only does this mean the agent is doing a lot of work, but also that this work implies communication with the servers involved. Optimisations in this area are indispensable to reduce expensive network calls. However, optimisations cannot be made without a plan for all actions to take. This yields in a contradicting situation where on the one hand the full plan needs to be expanded and on the other hand the costs of unnecessarily expanding the plan needs to be avoided. A conventional approach to query execution requires a full plan to be made before the actual execution starts, such that optimisations aimed at the full execution can be made. In the case of Armada, the tree-shaped lineage tree offers the opportunity to minimise the granularity of the plan gradually as the agent executes it. With the agent contacting servers to ask for the data, every time it receives a query plan back, it can update its query plan and do incremental optimisations during the execution phase.

Figure 8.9 depicts the architecture of the approach taken to obtain stepwise dynamic inlining. The behaviour of the Armada agent to possibly change its query plan during execution suggests it has to rewrite the currently running code. While in Armada this means that one instruction is replaced by a number of others (inlining) at runtime it imposes a serious administration burden to do so. Many references from and to values placed on the stack may need to be shifted in the current context. While an implementation like that has many potentials to cause random *crashes* in the implementation of MAL which was not designed to support this, a multi-staged solution was used as shown in the figure.



```
...
b := stub1();
io.print(b);
...
```

```
function stub1():bat;
  r := remote.exec("b1");
catch RedirectException re;
  r := armada.rewrite(re);
exit re;
  return r;
end stub1;
```

```
function stub1():bat;
  l := stub2();
  r := stub3();
  b := algebra.sunion(l, r);
  return b;
end stub1;
```

```
function stub2():bat;
  r := remote.exec("b2");
catch RedirectException re;
  r := armada.rewrite(re);
exit re;
  return r;
end stub2;
```

```
end stub3;
```

```
function b1():bat;
  r := bbp.bind("b");
  return r;
end b1;
```

```
function b1():bat;
  e := armada.redirect(b2, b3);
  raise e;
end b1;
```

```
function b2():bat;
  r := bbp.bind("b");
  return r;
end b2;
```

```
function b3():bat;
  r := bbp.bind("b");
  return r;
end b3;
```

Figure 8.9: Stepwise Dynamic Inlining in action.

Like in the analytic approach, the query of an agent is transformed such that requests for data in an Armada are made available in the plan. In the dynamic approach *stub functions* are used instead of `armada.bind` calls. The purpose of the stub functions is to implement dealing with redirects encountered at runtime. The function `stub1` tries to execute a function on the remote site to obtain the data. It is prepared to receive an exception which like a redirect contains enough information to create two new stub functions `stub2` and `stub3` and to redefine function `stub1` as a union of the result of the two new stub functions. After the rewrite the new function is called, which may iterate the same process again to obtain the data.

Dynamic Rewriting Going back to the abstract level of Figure 8.9, a box on a server is represented by a function that initially returns the data in the box as a BAT. The function remains at the server and is not meant to be shipped to any other site, or its implementation to become visible to others. This characteristic forces an agent to call the (remote) function when it wants to obtain the data from the box. Whether or not such call is cached as query plan, the real data being retrieved is always correct, as in, not out-of-date. When the box is chunked, its function is replaced by one that raises an exception which includes redirect information. An agent calling the function to retrieve the data's box, then receives the raised exception which allows to handle the redirection in the query execution.

From the agent's perspective the redirection exception raised by the remote function initiates a procedure to follow the redirection in the query. Since this situation occurs at run-time, a workaround to avoid rewriting the currently executing query plan is made through use of the aforementioned stub functions. Internally, when a new function is called, a new execution environment is used. For this reason, the soon to execute function can be created or modified just before execution. Also, functions can be overwritten, thereby "changing" the definition of the function for the next caller of that function. These ingredients are used by `armada.rewrite` to effectuate the redirect received from the server. First it creates two new stub-functions for the new boxes received via the redirect. These stub-functions are created by a template which simply tries to retrieve the box' data as BAT, or handles the redirection exception. Because those stub-functions are specific for a given box, the location (not shown in the figure) and name of the function to call are hardcoded in the stub-functions. Stub-functions cannot be reused, since they are overwritten when a redirect is encountered. Hence, hardcoding the remote function information in the stub

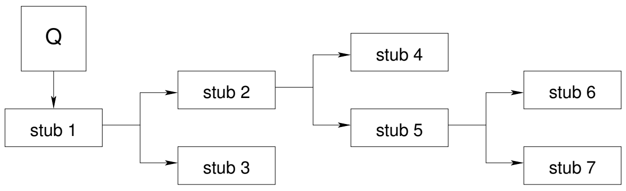


Figure 8.10: A trail of stub functions.

is not limiting in any way from a code and architecture point of view. Finally, `armada.rewrite` overwrites the function it was called from with a simple function which calls the two new stub-functions and returns the union of both BATs. Since execution of the original function where `armada.rewrite` was called is not affected, `armada.rewrite` calls the new function which overwrites the old stub and returns that result to the original stub-function, which is written in such a way that the data is finally delivered to the user's program. Note that the original stub-function `stub1` is created as part of the particular Armada initialisation on the system. It is part of the Armada catalog that enables availability of Armada BATs in the database system.

8.4.1 Caching

With stepwise dynamic inlining, each remotely known BAT from an Armada has a corresponding stub function on the agent. The stub functions are a necessity, but have a side effect of generating a cache on the agent. Each following use of `stub1` benefits from the already previously retrieved stub functions, as local access is cheaper and faster than constructing new stub functions from a remote redirect.

As the Armada grows, the agent has to store more stub functions. As a result, for each query the entire stub function tree has to be traversed. While still being local, these eventually long chains cause a lot of *function call overhead*. In addition, each query performed, has to execute the same path again, since they start at `stub1`. Figure 8.10 depicts such a trail of stub functions. Not surprisingly, such trail carefully follows the lineage trails of boxes. As such the same conditions of active and inactive stub functions hold. This means that stub-functions rewritten by `armada.rewrite` are not going to change and hence are eligible for inlining.

```
begin stub1():bat;
    _7 := stub6();
    _8 := stub7();
    _9 := algebra.sunion(_7, _8);
    _4 := stub4();
    _5 := _9;
    _6 := algebra.sunion(_4, _5);
    _1 := _6;
    _2 := stub3();
    _3 := algebra.sunion(_1, _2);
return _3;
end stub1;
```

Figure 8.11: A fully inlined `stub1` function.

Inlining Stubs Inlining the stub functions has the largest benefit when this is done in `stub1`, since each query using it then immediately accesses the fully inlined plan. Figure 8.11 depicts an inlined version of `stub1` in the situation depicted by Figure 8.10. Here, all possible stub functions are inlined. The resulting plan is flattened with the only functions not inlined, those stubs that represent active boxes, and hence may change in the future. Note that the inlining process has inlined the stubs in chronological order. As such originally `stub2` was assigned to variable `_1`. Further optimisations can reduce the size of the code by removing assignments such as `_1 := _6;` by propagating them through the code. This is commonly referred to as *alias removal*. The inlined plan in this state can be cached and reused for subsequent queries on the Armada BAT. Important detail here is that the inlining and optimisations are done on `stub1`, and not in the program that calls `stub1`. This is quite uncommon, but is the only way in which subsequent queries benefit from the work previously done. The contents of `stub1` may not be inlined into the calling program, since that would complicate matters in case one of the stub functions appears to be out of date.

Having rewritten stub functions being inlined in the top-level stub function, allows for generic optimisations to be applied only once, instead of for each query that uses the stub. Since active stub functions are not inlined, `armada.rewrite` which operates on those stubs, does not have to change the way in which it rewrites the stub functions. As a stub function gets rewritten, it effectively gets replaced without taking the original contents into ac-

```
...
a := bbp.bind("some_bat");
b := bbp.bind("another_bat");
...
p := bat.select(a, 10, 20);
q := bat.select(b, 40, 60);
...
t := algebra.join(p, q);
...
```

Figure 8.12: Example selection code.

count. This is no problem as long as the stub functions are not changed before `armada.rewrite` operates. The result of a rewritten stub-function can of course be inlined as well. For instance when `stub6` from Figure 8.10 gets rewritten, upon a next query it may get inlined into the `stub1` function. This scheme effectuates a self-reorganisation after a dynamic query refinement.

8.4.2 Volume Optimisation

In a distributed setting, optimisations on network costs always play a prominent role. In Armada, two different kinds of optimisation can be made in this respect. First, avoiding a call to a stub function yields in a full reduction of network communication. As side effect, it also avoids code expansion costs due to dynamic rewriting. Second, the data being retrieved can be minimised by pushing selections down into the remote server that holds the data. While this takes extra communication to retrieve only the selection, it potentially pays off against the number of data tuples that do not have to be shipped between the sites. For both optimisations it is necessary to have the *Armada functions* from the model. Without them, no information on what data is contained where is available, and hence no box can be assumed not to have the data. Encoding the functions in MAL can be done by assigning properties to the BATs returned by the stub functions. Properties are from a human code consumption point of view natural elements to tag objects with certain characteristics. Also for optimisers, for example minimum and maximum values as properties of a BAT are easy to consume and process.

```
tmp1{pmin=0,pmax=50} := stub2();
tmp2{pmin=50} := stub3();
tmp3{pmin=0} := algebra.sunion(tmp1{pmin=0,pmax=50}, tmp2{pmin=50});
b{pmin=5,pmax=20} := bat.select(tmp3{pmin=0}, 5, 20);
io.print(b{pmin=5,pmax=20});
```

Figure 8.13: An Armada chunk stub function using properties.

Code Optimisation Consider the example in Figure 8.12. There are no properties in the code, but it is not hard to imagine that a and b have undefined minimum and maximum values, whereas p and q have limits between $10, 20$ and $40, 60$, respectively. The final join is guaranteed to have an empty result, as p and q cannot have any tuples in common. End result is that the `bbp.bind` calls can be removed from the plan, thereby reducing actual work to do. Existing optimisers are already able to effectuate this, based on the previously mentioned code analysis. Reflecting this on the stub functions, inserting `select` calls on the result of the `remote.exec` calls would allow for the same optimisation without introducing an Armada specific optimiser. This contrasts the `armada.chunk` function used in the statical analysis approach.

Consider Figure 8.13 which depicts a selection made over a typical union of two stub functions. The figure shows the use of properties for the BATs in use. The possible minimum (`pmin`) and maximum (`pmax`) properties for the results of the stub functions are part of the stub function's definitions. Function `stub2` covers a range from 0 till (not including) 50. Since `stub3` has no upper limit on its range, only the minimum possible value (50) is encoded as property. In the generality of the property management, analysis of the property sets upon operations result in properties on return values. In the case of the `algebra.sunion` a "union" of both BATs theirs `pmin` and `pmax` properties results in the range of 0 till infinity. Of course the result of a `bat.select` call results in a BAT with `pmin` and `pmax` equalling the selection criteria. With all these properties available, a few conclusions can be made, when looking at the code from the bottom to the top. The selection on `tmp3` can possibly result in something, since the input BAT possibly holds data in the selection range. If this were not the case the `bat.select` statement could be replaced by an empty set. Such operation can yield in more statements becoming void, which are removed by the empty set optimiser.

Another optimisation based on the previous example is to reduce the size of the sub-results, in particular because they need to be fetched from a remote

```
tmp1{pmin=0,pmax=50} := stub2();
tmp4{pmin=5,pmax=20} := bat.select(tmp1{pmin=0,pmax=50}, 5, 20);
tmp2{pmin=50} := stub3();
tmp5{rows=0} := bat.select(tmp1{pmin=50}, 5, 20);
tmp3{pmin=5,pmax=20} := algebra.sunion(tmp4{pmin=5,pmax=20}, tmp5{rows=0});
io.print(tmp3{pmin=5,pmax=20});
```

Figure 8.14: Selection push-down applied on the example of Figure 8.13.

site. In Figure 8.14 the selection is pushed through the union operation such that the selection is done right after the stub functions over `tmp1` and `tmp2`. Due to the properties, it can be seen that the selection over `tmp2` is going to be empty, resulting in an empty set, indicated by the property `rows=0` on BAT `tmp5`. An effect of this knowledge to the empty set optimiser is that the union operation is now useless, and the entire operation can be skipped, with the result being just `tmp4`. The `bat.select` on `tmp1` can be removed as part of this, since the operation does not have any effect. In turn, the dead code optimiser detects that the `stub3` function call is needless, as its result is never used. This finally leads to avoidance of any network communication performed by function `stub3`.

Runtime Optimisation The set of regular optimisers work fine here on the inlined code, as shown before. An important problem, however, is that the `stub1` code cannot be inlined as we concluded before. Newly rewritten functions would not benefit from the optimisations, and optimisations applied to existing code possibly breaks for other queries. Regarless, selection optimisation as done above has the desirable effect of reducing network communication. Not inlining `stub1` makes the stub function a black box to the optimiser. Nothing can be done to it, the same situation that characterises a just rewritten stub-function, and selection optimisation therein. Temporarily inlining only works for the stub-functions representing an inactive box, the active ones can get re-written at runtime. Still, it means a temporary copy is made which is modified for the query at hand. Since the dynamic nature of the Armada query strategy forces a runtime based strategy, static (analytic) optimisers per definition fail to meet the requirements. Instead of them, runtime guards to avoid unnecessary work can provide generic avoidance of expensive network calls, at the expense of slightly higher execution costs. To do these runtime checks, more information is necessary at runtime in the stub functions. Starting with the user's original query again, we can use optimisers and static analysis to find if there

```
...
b := stub1(0, 10);
c := algebra.select(b, 0, 10);
io.print(c);
...
```

Figure 8.15: Optimisers can “push” the selection criteria through the stub functions.

is a selection made on the BAT retrieved by the stub function being called. This selection range then can be made an argument to the stub function call, such as in Figure 8.15.

The low and high values of the selection range are put as arguments of the `stub1` function, making the selection operation on `b` unnecessary. It is left here in the figure for explanation purposes. When no selection is made or its range cannot be determined, the default low and high values of `nil` are filled in which denote an unlimited range. A range for example cannot be determined when the selection is done using variables filled in at runtime. Though to a certain extent, also these variables can in some cases be used as arguments for the stub function. In some cases execution order may be changed such that the selection value is known in time for the stub function at runtime, but this may be too complicated to derive, or simply impossible. With the possible selection criteria the stub function can now use this information to possibly skip consulting remote servers that do not have the requested data. Instead of using properties that aid in statical analysis of the plans, the data coverage of the stub functions is now encoded in the code by means of the guards.

In Figure 8.16, a quick exit is encoded in a stub function to avoid performing the remote function call to retrieve the data. To not to disturb the calling code, an empty BAT is returned otherwise. Upon rewrite of the stub-function, the guards can be placed in the rewritten function as well, such as depicted in Figure 8.17.

With this code, it is no longer necessary to have any guards in the active stub functions, since they are not called when they cannot produce data matching the selection criteria. Note, however, that the selection range is now also given as argument to the `remote.exec` call. This allows the remote site to only return what is necessary for the selection. This is a further reduction of network traffic that comes for free now the selection range is pushed through all stub functions being called. New stub functions generated by the `armada.rewrite` call hence

```
function stub1(low:int, high:int):bat;
  barrier e := low > 10 || high < 0;
    r := bat.empty();
    return r;
  exit e;
  try;
    r := remote.exec("b1", low, high);
  catch RedirectException re;
    r := armada.rewrite(re);
  exit re;
  return r;
end stub1;
```

Figure 8.16: Runtime guards in active stub functions.

```
function stub1(low, high):bat;
  barrier e := low > 10 || high < 0;
    lb := bat.empty();
  opposite e;
    lb := stub2(low, high);
  exit e;
  barrier e := high < 10;
    rb := bat.empty();
  opposite e;
    rb := stub3(low, high);
  exit e;
  b := algebra.sunion(lb, rb);
  return(b);
end stub1;
```

Figure 8.17: Guards placed in an inactive stub function.

include `low` and `high` arguments and the respective guards. The `low` and `high` arguments are passed on to the `remote.exec` call. Because `stub1` is generated by the Armada initial ritual, it can contain the guards as shown before, such that calls to `stub1` in user code need no transformation to add the guards. This means that analytic compilers can do their job on the user's code up to the point where they know what the selection criteria are for the `stub1` function. From there the execution dynamically deals with the optimisations at runtime.

Summary

An Armada implementation requires more functionality from a generic database engine, than it can deliver via the SQL layer. To overcome this functionality shortage, we chose MAL, the algebraic MonetDB Assembler Language, as target for our next exploration of an Armada implementation.

Our implementation focusses on the columns present in the database, and by means of wrapper functions for those columns, Armada boxes are simulated. Since the functions can be changed, an operation is administered in such function by changing its code, and creating new functions representing the new boxes. The lineage in these functions is encoded by the call to other functions, and available to others by copying the functions.

We put some attention to optimisations in the query execution over an Armada. Compile time variants of MAL plans are able to reduce a lot of work, but they are hampered by the execution phase which can find a situation not accounted for. Instead, runtime variants are necessary to limit the communication with other sites to a minimum, by requiring only a single call to either get an answer or redirect.

Caching the MAL functions received from other sites allow to speed up the process by requiring less communication with other sites. Many optimisation efforts can be performed on the cached plans, resulting in a high benefit due to reuse.

With the final proposed implementation on the MAL language, we have shown that it is possible to implement an Armada agent conforming to the model. This implementation respects the autonomy of the involved sites, supports the decentralisation, and hence does not block the evolution of the cluster.

9 Conclusions and Future Work

9.1 Contributions

As the introduction of this thesis mentions, trends in computing change. Driven by requirements, ideals and most of all humans, our creation changes, and will change many times in the future. This thesis builds upon a trend that puts focus on the distribution of tasks over multiple machines. Scaling of performance is no longer seen as a viable option for a single, large machine. Instead, each partial machine that the entire system is made of, adds its own share to the performance of the system.

Also database management systems have been following this trend. For them, data growth and the processing thereof are simply growing out of bounds of a single system. However, as growth in terms of data and usage continues, the number of systems grows as well, causing an inevitable administration hurdle. To overcome this hurdle, a system has to manage itself to a large extent, alleviating the database administrators. This thesis deals with the autonomy implied by this self management in a distributed setting, which is expected to evolve where necessary.

The aim of this thesis is to be an exploration of autonomy, decentralisation and evolution in the context of database systems. Previous research in these three areas is mainly focused on decentralisation with autonomy as side-effect. In particular P2P systems which are primarily decentralised have been given a lot of attention. Autonomy is considered to go hand in hand with decentralisation, since a central controlling component is absent in decentralised systems. However, the amount of autonomy given to the participants of distributed systems is limited. Evolution has mainly been approached from a system internal angle.

The contribution of Armada extends autonomy to the level where participants are initiators in the system. Through this initiative, evolution of the entire system is achieved. Evolution, here, is not just an internal matter, but at the level of the global system, and can be seen as self-managing behaviour. The proposed Armada model decentralises by storing partial catalog information throughout the entire system.

The autonomy of the system enforces clients to play an active role in query resolution. This gives the client an unusual task, for the benefit of having the user controlling the query process. This is in particular useful using the incremental query evaluation that is a result of the Armada lineage tree. There are natural points where the query can be stopped or suspended, since data is divided in blocks. These blocks are defined by arbitrary functions, allowing full control over the characteristics of the data in a block.

9.2 Research Questions

In our exploration towards *autonomy*, *decentralisation* and *evolution* of database systems, we were lead by a general research question, consisting of four more specific questions. We first answer the four questions, followed by the answer to the general question.

9.2.1 In what way can we distribute data in a dynamically evolving system using site local decisions and avoid global site control?

The Armada model from Chapter 3 describes how data can be spread over multiple sites starting from a single one. Growth in this model is achieved by performing operations which involve sites being added to the system. The operations are performed by the local site that was triggered to do so by e.g. a resource limitation. Such site is an initiator to resolve a local problem, on its own. Other sites in the system need not to express agreement, or be informed of the performed operation. Obviously, this means only those sites which are not directly involved in the operation. This independence of sites for their local decisions is due to the trail administration of the Armada model. The trails store pointers to other sites, that may or may not be up-to-date. Upon creation, history information is inherited from the local site, allowing each site to refer back to predecessors. Since the pointers may not be up-to-date, when following them it has to be accommodated for that they are out of date. This leads to locally performed operations to become visible when they are referenced. Since operations always redefine where data is that used to be on the original site, and hence when following the trails, one never suddenly ends up in the wrong place.

The lineage trails of the Armada model are a way to administrate the whereabouts of data, without a central catalog scheme. Dynamically evolving systems benefit from this scheme, since they can expand when and where necessary without being hampered by costs of e.g. global updates and checking for their consistency. The use of arbitrary functions in operations, gives full control to the local site performing the operation.

9.2.2 What is the role of application clients in an autonomous, distributed database management system?

To maintain their high level of autonomy, servers do not perform work for which others are responsible. In practice this boils down to the clients in the system acting as directors in query execution. This unusual role for a client is discussed in Chapter 5. While it is intensive to perform all actions during query resolution, agents can be of help to the client. Agents can perform simple roles, such as following redirects and assembling a query result, if the client desires no explicit control over that process.

The process of query resolution can become expensive when an Armada grows large. Since clients, whether or not helped by agents, need to traverse the tree, the bigger it becomes, the more steps are possible. In Chapter 6 this process of localisation is further explored. To reduce the number of steps per query, the client or agent best uses a cache of visited trails for subsequent reuse.

9.2.3 How can incremental scalability become a natural component in an evolving system?

The operations from the Armada model described in Chapter 3 form the basis for a system to evolve. By means of these operations it can grow or shrink over time. This behaviour is already part of the design of the model and hence a natural component. The operations, however, need the functions to specify how exactly the operations need to be performed. In Chapter 4 we identified a number of functions and described their effect both separate as well as in combination with each other. In addition we looked at how functions can be chosen automatically for the situation at hand, aiming at a naturally self-evolving system.

9.2.4 To what extent are existing common techniques to manage a catalog sufficient to support autonomy, decentralisation and evolution?

The proposed model in this thesis breaks with traditional methods. Nevertheless we tried to simulate it using the SQL language in Chapter 7. While querying could work on existing database systems using this SQL approach, the autonomy considerations of the Armada model have to be ignored, since no client redirection is possible. The remote querying facilities are also not standardised and not necessarily sufficient to perform Armada operations. The SQL approach also falls apart with updates because of limitations of the used views.

However, while existing database systems are incapable of supporting Armada when implemented in the SQL language, it is not impossible to implement Armada. In Chapter 8 an approach on a deeper level towards the database kernel is explored. It shows that on those lower levels there is more freedom which allows to make a sufficient implementation. If one is willing to omit user interaction during query resolution, the interface can even use the SQL language, because the Armada implementation is below it, and invisible for the user.

9.2.5 How to support a continuously evolving database management system consisting of autonomous sites and a decentralised catalog?

In the previous sections we answered the four specific research questions. Now we are able to answer the main research question of this thesis.

Our exploration in the area of autonomous distributed database systems has resulted in a model which describes decentralised autonomous evolution in an environment of computers and schema-based data. The model handles the trade-off between full replication and centralisation of the catalog, by well targeted partial replication of the catalog meta information. This way, the catalog is distributed over the entire system as part of its own evolution process.

An important aspect of the model, is its assumption of local autonomy. It is this autonomy that allows the local sites to initiate a next step in the evolution of the system. But the autonomy also affects the users of the system. We have explored the effects of this autonomy on the clients in the system, in particular how well they can perform queries. We can conclude that it is possible to do so, even though not with standard techniques available today.

A key to the self-evolution of the system is in the way functions, as part of the Armada model's operations, are created. The functions define how the data is distributed, and when created by local sites based on local conditions, also form the driving power behind evolution of the system as a whole.

9.3 Future Work

With our exploration we have scratched the surface of a decentralised, autonomous and evolving world. We did not round the Cape, but fortunately did not ask to keep on trying until the youngest day either. Docked at the Table Bay, we leave a lot of work left for others to pick up and continue with. The following areas are of particular interest.

Functions The functions that back up the operations are only partially addressed in this thesis. Not only have we largely ignored the cloning and combining functions, but also the chunking functions have been marginally addressed with only a range function implementation. Different functions have different behaviours and they may be more beneficial under certain workloads.

In addition the conditions under which decisions for cloning and chunking are made, are a separate topic potentially borrowing from the artificial intelligence area when seen from a multi agent system angle.

User Input One of the interesting properties of the Armada model is the close user interaction. This allows for handling very specific user demands. We have left this mainly untouched, but we can think of applications where this control is beneficial. A few ideas are skipping certain boxes in query resolution, aborting the execution, or suspending it for a long time.

Performing such interaction requires changes to existing user interfaces though. To benefit optimally from this interaction the user needs to be provided with many meta data over the system and the state of the query resolution process.

MAL Architecture The approach described in Chapter 8 has been validated against the system in question. However, the missing parts to make it a working system, still need to be implemented. The described `remote` module, and the Merovingian service described in Appendix A are blocks for an actual implementation, but they need to be put into action still. The most work is in the optimiser architecture that plugs in the armada behaviour on top of an SQL plan, as to make it work from a user's perspective. Of course the previously mentioned user interaction considerations are also a possible design and implementation adventure.

Physical Allocation While we described how sites can be selected from a pool, the way in which this pool is allocated and maintained leaves some questions. The physical implementation of "plugging in machines" still requires some administration, and a machine that is removed should be properly unregistered by making sure its data is cloned to another machine.

The process of finding a new site to host a box on, is rarely a case of choosing one site from a pool of homogeneously typed machines. Instead, a mix of fast, slow, multi-way, large and small systems in terms of processing power and data storage capacity is more likely to be in the pool. Next to these properties, also the network link that connects the sites is an important factor in their performance. These properties ask for a careful cost/revenue analysis, based on predicted loads and functionalities. Some machines may even announce their life time in the system, making them only useful for e.g. facilitating a burst.

A MonetDB Clusters

A.1 Merovingian: MonetDB as a Service

The MonetDB software platform has traditionally always been centred around a console based application, the *mserver*. Interaction via the console with the database server is made using the kernel language, which traditionally is incomprehensible for normal users. This setup, where the console is always present greatly aids developer interaction and debugging, but throws up a barrier for normal and business users. Normally, a database server runs as *daemon* (server) process in the background, responding to client requests and writing log messages to some file on disk. Stopping the daemon is done by sending it a termination signal, instead of interacting via the console to quit the server. For a successful Armada deployment, non-interactive starting and stopping of database servers is essential to keep maintenance low.

Because changing the characteristics of *mserver* results in a lot of resistance by its core developers whose strong habits rely on the console based interaction, a wrapper-like approach was chosen to obtain the above described desired behaviour for the MonetDB database system. This wrapper, called *Merovingian*, has next to the features of a daemon some options that make it a core component of an Armada deployment. Merovingian here does not operate on its own, the *Sabaoth* layer was added to help Merovingian on managing a local database farm.

A.2 Merovingian's Architecture

Figure A.1 shows a cluster of two sites, equipped with Merovingian instances. Focussing on a single site for now, a single Merovingian can manage multiple Mservers through the Sabaoth disk-based administration. The role of Mero-

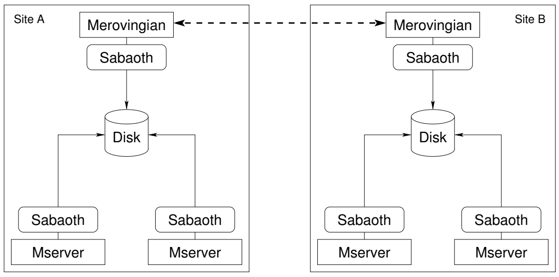


Figure A.1: The architecture of an Armada cluster and its core components.

vingian is threefold. First, it provides daemon functionality for an Mserver. Second, Merovingian (re)starts an Mserver once a client requires a connection to it. Third, it handles discovery and cooperation with remote sites also using Merovingian. The first two roles go hand in hand. An Mserver started by Merovingian, remains under its umbrella and hence can be monitored. Hence, Mserver emitted messages can be caught and logged. Upon shutdown, Merovingian first shuts down the Mserver before it shuts down itself. The ability of Merovingian to restart an Mserver comes in handy when an Mserver has crashed or was shut down temporarily for maintenance.

To perform its job, Merovingian acts as a server to a connecting client. This means that a client actually connects to Merovingian instead of an Mserver. This is easily achieved by having Merovingian running on the default port for an Mserver. Once a client has made a connection, it informs the server about what database it is looking for, as usual. Merovingian looks up this database and depending on its state it (re)starts the corresponding Mserver if necessary. After making sure that the Mserver is running, Merovingian redirects the client to the Mserver either using a redirect response or by creating a proxy to it. A redirect response causes a client to disconnect and follow the directions given by the server, a proxy causes the client to restart its login ritual. In case of a redirect, Merovingian is not in between client and server, thereby avoiding to become a bottleneck, but not all network setups support this due to firewall configurations or routing issues.

A.3 Sabaoth's Disk Administration

When Merovingian needs to look up the state of an Mserver, it actually consults Sabaoth. To achieve maximum independence and platform independent support, Sabaoth uses the local disk to store information about the state of an Mserver. Sabaoth is no actual process itself, instead it is a set of functions that read from and write to the disk upon request. The actions that Sabaoth supports for storage and retrieval are numerous. It keeps track of start and stop information and the crash information that is implicitly encoded therein. Each database records how it can be reached, via the available connections administered, next to the available scenarios for such connection, such as e.g. the SQL language. Lastly, Sabaoth can list all known databases in the local database farm, including the properties of those databases.

Using all of this information provided by Sabaoth, Merovingian can find out if a database exists or not, if it is already running or not and when both are true, how to connect to it. The disk administration of Sabaoth is straight-forward based on separate files, placed in the database directory of the respective database.

Uplog The `.uplog` file consists of timestamps representing start and stop times of the server. Sabaoth only appends to this file, hence the risk of corruption is low. A start time is followed by a tab character, whereas a stop time is followed by a newline character. As such, the `.uplog` file looks like a two column data sheet on a database that has no crashes. As soon as a database crashes, no stop time is written to the log, and hence the next start of the server follows the previous start time. For both the human eye and a program it is easy to see a crash has occured in such case.

Connections All sockets that are opened by a server to listen for connections, are registered by Sabaoth in the `.conns` file. It contains one or more URLs that point to the database, e.g. `mapi:monetdb://localhost:50001/`, which can directly be used to connect or redirect to that database. Since an Mserver calls the respective Sabaoth function to register an available connection only when it has successfully completed opening that connection, Merovingian can reliably redirect a client to a just started database once the Mserver has registered available connections through Sabaoth.

Scenarios A scenario is a specific language mode that can be used to communicate to the server. Each Mserver can have a different set of scenarios loaded and hence available. The `.scens` file contains all scenarios available, one scenario per line. Typically, the file contains `sql` and `mal` entries, for availability of the SQL language front-end and the MAL kernel language console.

A.4 Remote Databases with Merovingian

Merovingian is considered to be the entry point for clients to the MonetDB Database Server on a particular system. As such, in distributed scenarios Merovingian can be used to represent the "global" knowledge that is necessary to find another database in the environment. *How* Merovingian does this, is in principle unrelated to the fact that it *does*. In other words, a client comes to Merovingian, assuming it gets an appropriate redirect or error when the database does not exist. The implementation used in Merovingian to know about foreign databases should just satisfy the assumption of the client. Currently Merovingian uses a simple broadcast based implementation. Future versions might use a DHT based approach if the broadcast approach is proven not to be sufficient. To redirect a client to a foreign database, Merovingian needs a Sabaoth like structure containing at least the database name and its connection URL. Since most tools relying on Sabaoth assume it only administers local databases, appending the remote database structures to it results in conflicts. This is best illustrated by the properties of such remote database. The properties available in Sabaoth about databases need not to be available for a foreign database. Retrieving the information may be expensive. For these reasons, the information only makes sense for Merovingian. Each Merovingian publishes information about its databases either push or pull based to others with a modified connection URL. Instead of associating the local connection URL to a published database, the connection URL of the Merovingian publishing it is used. This scheme allows the foreign Merovingian to start a requested database on the fly when necessary, as normal.

Broadcasts In the broadcast implementation of Merovingian, each starting Merovingian broadcasts a list of available databases using `ANNC` "announce" messages. Other Merovingian processes that receive those messages append these databases to their local administration of remote databases. Next, a starting Merovingian sends a `HELO` "hello" message which other Merovingian processes respond to by announcing their available databases. This way, a joining Merovingian publishes its existence and an exchange of available databases throughout the network is performed.

The broadcast can be limited to a certain subnet, as to reach only a selective audience, or IPv6's advanced multicasting features can be used to control the reach of broadcast messages. For simplicity all communication is done using connection-less UDP packages, which allow broadcasting and are very cheap. Loss of messages does in principle not harm the system as a whole. Considering in most cases the cluster runs over a local network, it is unlikely packages get lost at all.

Time-To-Live New databases can be added, or others removed. Merovingian itself does not notice this until a client asks for such database. Hence the only way of finding new databases, is by periodically consulting Sabaoth to retrieve the current list of available databases. This way it is easy to detect a new database, but hard to detect a removed database. For this a list of the previous consult has to be kept. An alternative solution using a time-to-live is used by Merovingian. Each announcement of a database gets a time-to-live (TTL) from the announcing Merovingian. Each Merovingian that receives this announcement stores the TTL, and frequently checks all known remote databases whether their TTL has expired. If so, the remote database is dropped. To prevent a remote database from being dropped, its originating Merovingian has to "renew" the TTL, which is done by a re-announcement of the same database, but with an updated TTL. Each Merovingian periodically announces its local databases this way, a little earlier than the TTL expiration time. This way it is avoided that a database becomes unavailable for a short period around TTL expiration. Local databases that are removed, are not re-announced this way, and hence due to the TTL expiring they are also removed from remote Merovingian processes, be it with a delay. The same delay is encountered for new databases to become available at remote sites. However, it is not common that databases are added or removed. For environments there this is the case for some reason, the TTL time can be set relatively low to increase the responsiveness here.

With each Merovingian periodically re-announcing, the cycle in which this is done is based on the TTL time and the startup time of the Merovingian. This is important for the load on the network, since broadcasting is used. If all Merovingian processes would re-announce at the same time, a flood of messages would occur and package loss or even worse a Denial of Service (DoS) happen. While this seems unlikely to happen, the `HELO` message could synchronise all Merovingian processes when they would reset their TTL when announcing in response of the hello. Hence, the announce messages in response to a hello message are followed by a re-announce within the TTL expiration time, just to avoid repetitive mass announcements. To avoid the same effect as response to the hello message, each Merovingian takes a random delay before sending the announcement.

Finally, when a Merovingian is shut down, it sents a `LEAV` message for each local database. This is a little “service” to speed up the process of removal of remote databases when they are no longer valid. Of course when this step would be skipped, a TTL expiration would clean up eventually.

A.5 Interoperability

Even though this is a very simple way of publishing availability of data sources, it does allow for processes other than Merovingian to publish its own existence. In a Merovingian only environment, all participators administer foreign databases. This symmetry is however, not a requirement, and as such a compatible process (such as an old version of Mserver) that is not compatible with Sabaoth could publish itself to Merovingian with a minimum effort. In reality this is an issue where MonetDB v4 has support for XQuery, unavailable in MonetDB v5. The former, however, cannot be managed by Merovingian which is designed for the latter. By changing the MonetDB v4 server slightly to publish itself using the simple message mentioned upon a broadcast of a Merovingian, its XQuery facilities can be made available through Merovingian, without its explicit control over it. This doesn’t provide the support of Merovingian for MonetDB v4 of course, but it helps to make it available during the transition period. Another example would be specialised “routers” that connect Merovingians on two different configurations (ports) to each other by implementing a cross. Since full URLs are stored, the eventual client connection is not bothered by the configuration difference.

A.6 Merovingian and Armada

The database names published, are used as is by Merovingian, e.g. when the connection url for a foreign database is `mapi:monetdb://myhost/` and its published database `mydb`, then the redirection url that Merovingian uses for this database is `mapi:monetdb://myhost/mydb`. Internally to Merovingian, however, the foreign database is known as `mydb`, exactly like as it was published.

This scheme is likely to result in conflicts; it is not hard to imagine two or more Merovingians to have the same database name in a default setting. For each Merovingian, however, the Sabaoth administration is consulted before any foreign published databases are searched. Duplicate database names in the foreign adminstration of a Merovingian are simply stored, but the first found entry is used. Since this most likely depends on the order in which foreign Merovingians published their databases, this may differ per Merovingian and even per run. The alternative of making each database name unique results in those databases not being able to be found under their original name any more. For this reason, the uniqueness of database names is left to the actual users of the cluster of Merovingians. In an Armada setting this is not an issue at all. Databases are made as part of the initialisation ritual of a certain node. For that purpose it is more than useful to create a database that has a unique name in the cluster by e.g. combining the node name and its local sequence number. The "service discovery" functionalities of Merovingian are for this case sufficient.

Bibliography

[1] E. H. L. Aarts et al., editors. *First European Symposium on Ambient Intelligence (EUSAI)*, volume 2875 of *LNCS*, 2003.

[2] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[3] S. Agrawal et al. Database tuning advisor for microsoft sql server 2005. In *VLDB*, 2004.

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[5] H. Balakrishnan et al. Retrospective on aurora. *VLDB Journal*, 13(4), 2004.

[6] P. Bernstein. Applying model management to classical meta data problems, 2003.

[7] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, UVA, 2002.

[8] P. A. Boncz and C. Treijtel. AmbientDB: relational query processing in a P2P network. In *International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P) (co-located with VLDB 2003)*, volume 2788 of *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence (LNCS/LNAI), © Springer-Verlag*, Berlin, Germany, September 2003. Also available as CWI Technical Report INS-R0306.

[9] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with dbcache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.

[10] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. Sql anywhere: An embeddable dbms. *IEEE Data Eng. Bull.*, 30(3):29–36, 2007.

[11] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.

[12] S. Ceri and G. Pelagatti. *Distributed databases principles and systems*. McGraw-Hill, Inc., New York, NY, USA, 1984.

[13] S. Chaudhuri and V. Narasayya. AutoAdmin “what-if” index analysis utility. pages 367–378, 1998.

[14] S. S. Chawathe. *Managing change in heterogeneous autonomous databases*. PhD thesis, Stanford, CA, USA, 1999. Adviser-Hector Garcia-Molina.

[15] J. Chen, G. Soundararajan, M. Mihailescu, and C. Amza. Outlier detection for fine-grained load balancing in database clusters. In *ICDE Workshops*, pages 404–413, 2007.

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[17] B. Cook, S. Babu, G. Candea, and S. Duan. Toward self-healing multitier services. In *ICDE Workshops*, pages 424–432, 2007.

[18] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6), 1992.

[19] K. Douglas and S. Douglas. *PostgreSQL, Second Edition*. Sams, 2005.

[20] M. H. Dunham et al. A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2(2), 1997.

[21] M. J. Franklin et al. Design considerations for high fan-in systems: The hifi approach. In *CIDR*, 2005.

[22] S. Gancarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel processing with autonomous databases in a cluster system, 2002.

[23] H. Garcia-Molina and D. Barbará. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[24] G. Gardarin. Iro-db : A distributed system federating object and relational databases, 1995.

[25] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979.

[26] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services.

[27] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2):102–111, 1990.

[28] J. Gray. An approach to decentralized computer systems. *Software Engineering*, 12(6):684–692, 1986.

[29] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do For Databases, and Vice Versa? In *WebDB*, Santa Barbara, CA, USA, 2001.

[30] F. Groffen, M. L. Kersten, and S. Manegold. Armada: a Reference Model for an Evolving Database System. In *Proceedings of Datenbanksysteme in Business, Technologie und Web*, Aachen, Germany, Mar. 2007.

[31] F. Groffen, M. L. Kersten, and S. Manegold. Optimising Client Accesses within Armada. In *Third Workshop on Dependable Distributed Data Management*, Nuremberg, Germany, Mar. 2009.

[32] L. M. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[33] L. M. Haas, P. G. Selinger, E. Bertino, D. Daniels, B. G. Lindsay, G. M. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. F. Wilms, and R. A. Yost. R*: A research project on distributed relational dbms. *IEEE Database Eng. Bull.*, 5(4):28–32, 1982.

[34] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, 1985.

[35] M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM '07: Proceedings of the ACM first Ph.D. workshop in CIKM*, pages 109–116, New York, NY, USA, 2007. ACM.

[36] S. E. Hudson and R. King. Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. Database Syst.*, 14(3):291–321, 1989.

[37] R. Huebsch et al. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR*, 2005.

[38] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: a balanced tree structure for peer-to-peer networks. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[39] JBoss-Inc. Hibernate. `http://www.hibernate.org/`, 2007.

[40] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In *SIGMOD Conference*, 2002.

[41] J. S. Karlsson. *Scalable Distributed Data Structures for Database Management*. Ph.D. Thesis, Univ. Amsterdam, Amsterdam, The Netherlands, December 2000.

[42] J. S. Karlsson and M. L. Kersten. Omega-storage: A Self Organizing Muliattribute Storage Technique for Large Main Memories. In *Australasian Database Conference*, 2000.

[43] R. King, M. Novak, C. Och, and F. Velez. Sybil: Supporting heterogeneous database interoperability with lightweight alliance. In *Next Generation Information Technologies and Systems*, pages 0–, 1997.

[44] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[45] W. Litwin et al. Lh*rs: A highly available distributed data storage. In *VLDB*, 2004.

[46] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, 1986.

[47] R. MacNicol and B. French. Sybase iq multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[48] S. Madden et al. Tinydb: an acquisitional query processing system for sensor networks. *ACM TODS*, 30(1), 2005.

[49] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM.

[50] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*.

[51] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[52] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems*, 2002.

[53] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal: Very Large Data Bases*, 6(1):53–72, 1997.

[54] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. *SIGMOD Rec.*, 21(2):361–370, 1992.

[55] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, pages 633–644, 2003.

[56] Y. Petrakis et al. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. In *DBISP2P*, 2004.

[57] J. Rao et al. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, 2002.

[58] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[59] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Proc. IFIP/ACM Middleware*, 2001.

[60] B. Schwartz, P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, and D. J. Balling. *High Performance MySQL, 2nd Edition*. O'Reilly, 2008.

[61] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. of the 14th International World-Wide Web Conference*, pages 33–42, Chiba, Japan, may 2005.

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

[63] M. Stonebraker. The Case for Shared Nothing Architecture. *Database Engineering*, 9(1), 1986.

[64] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[65] M. Stonebraker et al. Mariposa: A New Architecture for Distributed Data. In *IEEE 10th International Conference on Data Engineering*, 1994.

[66] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[67] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092. VLDB Endowment, 2006.

[68] M. Tamer Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice Hall, 1999.

[69] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[70] M. van Steen. Towards very large, self-managing distributed systems: Extended abstract. In *OPODIS*, pages 3–6, 2003.

[71] J. Veijalainen and R. Popescu-Zeletin. Map: an open multidatabase system architecture. In *EW 3: Proceedings of the 3rd workshop on ACM SIGOPS European workshop*, pages 1–4, New York, NY, USA, 1988. ACM.

[72] W. Vogels. Data Access Patterns in The Amazon.com Technology Platform, Keynote Speech VLDB, 2007.

[73] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 2002.

[74] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed Segment Tree: Support Range Query and Cover Query over DHT. In *5th IPTPS Workshop*, 2006.

[75] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

Summary

In a world where data usage becomes more and more widespread, single system solutions are no longer adequate to meet the data requirements of today. No longer one monolithic system, but instead a group of smaller and cheaper ones have to manage the workload of the system, preferably as stable as the large single systems currently in use.

The ultimate goal is to have a self-managing and self-maintaining cluster of machines that just needs maintenance in terms of physically adding and removing hardware every once in a while, to cope with changed requirements. Close to this objective are *Peer-to-Peer* (P2P) systems, which are well-known on the internet, and quite effective in distributing data over the network. However, these systems typically distribute only certain data over the network as side-effect of certain user demands.

This thesis explores the landscape of self-managing database systems. It takes autonomy, decentralisation and evolution as starting point for this exploration. Autonomy of an individual system allows how much a system is able to control itself, and make decisions for itself that put it in a better position, for instance by temporarily refusing to do work for others. This self-regulation allows for evolution of the entire system, where individual components work towards a new structure of the system that better matches the current requirements. Such approach leads to decentralisation, as there is no hierarchy since all systems are autonomous.

At the heart of this thesis is the Armada model which describes a method to distribute relational data, as found in typical database systems, over a cluster of machines. The model takes autonomy, decentralisation and evolution as starting points, resulting in a distributed administration. Since the administration is not managed in a single location, this way local systems can use their autonomy and change the administration for the part they are responsible for. Each system can do this without harming any of the other systems, thereby supporting evolution. Because this gives each system a large degree of freedom, they can even choose how to perform for example a split of the data, using the right methods to reach the required goal, if they deem this necessary.

A consequence of having autonomous systems in a cluster is that users of the system have to face systems that refuse to do work on their behalf. This translates into an active client model, where clients are responsible for the execution of their own queries. This can be intensive and unfriendly for a human user. Fortunately it is possible to automate a lot of necessary work in an agent that works on behalf of the user, by communicating to the systems. However, this comes at the price that this way agents remove the possibility for the user to influence the execution process, such as stopping the execution after a review

of intermediate results.

Agents that work on behalf of a user, looking for data in an Armada tree, need to hop around the cluster from system to system. The more hops an agent makes, the longer it takes, and hence the lower the performance of query execution. It is beneficial if the agent can reduce the number of steps it has to make, which it can do by caching information on the whereabouts of data it encounters when searching. The next time the agent needs to handle a query, it can then first consider its cache to see if it can directly go to the right site, or one nearby. In practice this allows an agent to quickly reduce the number of hops it has to make per query.

It is possible to map the Armada model to SQL, using *views*. This way, each individual data block can be represented by a view, that points to the right table, or when no longer existing, the replacement tables. This way an ordinary query can become a query over a large amount of tables through these views. While this works fine for expansion of the database, as well as querying the data within it, updating or inserting data is a problem, since the current SQL implementations do not, or not sufficiently, support updates on views, which in the Armada case can be complicated. Hence this approach turns out to be of limited use.

To solve the above problem, an Armada implementation deeper into the database system is necessary, such as at the MAL level of the MonetDB database. On this level, which is directly on top of the core engine, there are many degrees of freedom that allow to do more complex operations and optimisations. On this MAL level, an Armada system that supports reads and writes can be implemented.

Samenvatting

Sla een willekeurig automatisering blaadje open, en de hedendaagse kreten die “de trend” heten komen u tegemoet. Waar eens de GRID technologie het helemaal was, wordt u nu een SOA toegewenst. Of zo’n *Service Oriented Architecture* nu echt het einde of een aandoening is, het blijft onvermijdelijk om te concluderen dat grote, eenzame mainframes uit de mode zijn.

Niet langer één groot en zwaar systeem, maar in plaats daarvan een of meerdere clusters van kleinere en vooral goedkopere machines die samen de klus moeten klaren. Ook database systemen moeten eraan geloven, om overweg te kunnen met een cluster van machines, terwijl ze ook nog fatsoenlijk en betrouwbaar werk blijven leveren.

Is het niet de droom van elke computerfreak om een een cluster van computersystemen te hebben dat zichzelf onderhoudt? Een cluster waar enkel nieuwe hardware in hoeft te worden geschoven, om aan de nieuwe opslag- en verwerkingshonger te kunnen voldoen? Zo’n systeem is nu nog een utopie voor de database wereld, maar er zijn al stappen in die richting gezet. Het meest in het oog springend zijn de zogenaamde *Peer-to-Peer* (P2P) systemen. Deze internet brede systemen zijn afgezien van hun populariteit in de media in staat om data te verspreiden en lokaliseren zonder een centraal besturend medium. Maar in hoeverre doet het systeem dit nu eigenlijk zelf? Doorgaans is dit slechts een neveneffect van een bepaalde behoefte van mensen.

Dit proefschrift is een verkenningstocht in de richting van een zelf gedreven systeem. Onder de vlag van Armada, zijn autonomie, decentralisatie en evolutie als een combinatie onderzocht. De mate van autonomie van een computer systeem bepaald in grote mate in hoeverre het systeem zichzelf kan onderhouden. Een autonoom systeem kan zelf beslissingen nemen over zijn toestand, en bijvoorbeeld werk van anderen weigeren. Wil een systeem zelf evolueren naar een beter aangepast systeem, dan zullen de systemen waaruit het bestaat hun verantwoordelijkheid moeten nemen. Die insteek leidt vanzelf tot decentralisatie, waarbij geen machtshiërarchie bestaat aangezien elk subsysteem zijn eigen autonomie waarborgt.

Het Armada model, dat aan de basis van dit proefschrift staat, beschrijft een methode om data, zoals die typisch in een database gevonden wordt, over een cluster te distribueren. Het model speelt in op de behoefte aan autonomie, decentralisatie en evolutie door de feitelijke distributie administratie ook gedistribueerd op te slaan. Juist daardoor kunnen lokale systemen hun autonomie gebruiken om een een bepaalde operatie uit te voeren – meestal ter uitbreiding van het systeem. Immers, omdat de administratie gedistribueerd is, en wel precies naar de plaatsen waar ook de feitelijke data zich bevind, kan een systeem geheel onafhankelijk een operatie uitvoeren, zonder daarbij het totale systeem (deels) onbruikbaar te maken voor anderen. De vrije keuze van bijvoorbeeld een opsplitsfunctie stelt het betreffende systeem nog meer instaat zijn autonomie te beoefenen en een functie te kiezen met de eigenschappen die precies passen bij het beoogde doel.

Door de autonomie van de systemen in het cluster, ontstaat er een situatie waarin gebruikers worden geacht zelf actief met hun verzoeken bezig te zijn. De systemen zullen geen werk doen wat zij niet strikt hoeven te doen, met alle gevolgen van dien voor de gebruikers. Omdat dit overwegend simpel, maar behoorlijk intensief kan zijn is het mogelijk om zogenaamde agenten te gebruiken die als tussenlaag tussen de feitelijke gebruiker en het systeem zit. Dit is uiteraard prettig voor de gebruiker, maar sluit een mogelijkheid uit waarin de gebruiker de uitvoering van zijn verzoek wil beïnvloeden, zoals voortijdig afbreken na beschouwing van tussentijdse resultaten.

Een agent die in een cluster opzoek is naar de juiste data moet stappen maken. Des te meer stappen er gemaakt worden, des te lager de uiteindelijke prestaties. Elke stap betekend een verbinding met een ander systeem uit het cluster, en het maken van zo'n verbinding is duur. Het daarom belangrijk dat de agent voorgaande stappen niet direct vergeet, maar tot een bepaald punt onthoudt. Hiermee kan in de praktijk grotendeels stappen vermeden worden, omdat direct teruggevallen kan worden op de onthouden stappen. Uiteraard komt dit de uiteindelijke prestatie ten goede.

Om het Armada model af te beelden op de reguliere SQL taal, kunnen de *views* uit deze taal gebruikt worden om de verschillende data blokken te representeren. Op die manier ontstaat een aaneenschakeling van tabellen via views. Deze aanpak werkt in principe voor het reconstrueren van de data, maar toont gebreken wanneer data toegevoegd, of gewijzigd moet worden. Met name het niet kunnen bijwerken van data door een view gerepresenteerd maakt de aanpak ongeschikt.

Een manier om hiervoor genoemde problemen op te lossen is door op een dieper niveau in een database systeem het Armada model toe te passen, zoals bijvoorbeeld het MAL niveau in de MonetDB database. Dit niveau speelt op het niveau van de database kern, net boven de eigenlijke programma code. Op dit niveau zijn vele vrijheidsgraden die wijzigingen en toevoegingen op of aan de data mogelijk maken. Ook is het hier mogelijk om optimalisaties door te voeren door betere kennis over de uitgevoerde plannen.

SIKS Dissertation Series

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UvA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupé (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1	Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
2001-2	Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
2001-3	Maarten van Someren (UvA) Learning as problem solving
2001-4	Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5	Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6	Martijn van Welie (VU) Task-based User Interface Design
2001-7	Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8	Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9	Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10	Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11	Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01	Nico Lassing (VU) Architecture-Level Modifiability Analysis
2002-02	Roelof van Zwol (UT) Modelling and searching web-based document collections
2002-03	Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
2002-04	Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05	Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06	Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07	Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08	Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09	Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
2002-10	Brian Sheppard (UM) Towards Perfect Play of Scrabble
2002-11	Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12	Albrecht Schmidt (Uva) Processing XML in Database Systems
2002-13	Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications
2002-14	Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15	Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16	Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications
2002-17	Stefan Manegold (UvA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UvA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UvA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UvA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UvA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UvA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UvA)
Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UvA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramírez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UvA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UvA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20	Rex Arendsen (UvA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.
2008-21	Krisztian Balog (UvA) People Search in the Enterprise
2008-22	Henk Koning (UU) Communication of IT-Architecture
2008-23	Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia
2008-24	Zharko Aleksovski (VU) Using background knowledge in ontology matching
2008-25	Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
2008-26	Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
2008-27	Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design
2008-28	Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks
2008-29	Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
2008-30	Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
2008-31	Loes Braun (UM) Pro-Active Medical Information Retrieval
2008-32	Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
2008-33	Frank Terpstra (UvA) Scientific Workflow Design; theoretical and practical issues
2008-34	Jeroen de Knijf (UU) Studies in Frequent Tree Mining
2008-35	Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009

2009-01	Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models
2009-02	Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques
2009-03	Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT
2009-04	Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering
2009-05	Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
2009-06	Muhammad Subianto (UU) Understanding Classification
2009-07	Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08	Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09	Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems
2009-10	Jan Wielemaker (UvA) Logic programming for knowledge-intensive interactive applications
2009-11	Alexander Boer (UvA) Legal Theory, Sources of Law & the Semantic Web
2009-12	Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services
2009-13	Steven de Jong (UM) Fairness in Multi-Agent Systems
2009-14	Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15	Rinke Hoekstra (UvA) Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16	Fritz Reul (UvT) New Architectures in Computer Chess
2009-17	Laurens van der Maaten (UvT) Feature Extraction from Visual Data
2009-18	Fabian Groffen (CWI) Armada, an Evolving Database System