Incremental Data Distribution on Internet-Based Distributed Systems: A Spring System Approach

by

Catalin Visinescu

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2003

©Catalin Visinescu 2003

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Catalin Visinescu

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Catalin Visinescu

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Efficient data distribution is critical to enable cost-effective performance and high availability of data for applications or services. Many replication and allocation strategies were proposed for data distribution in traditional distributed DBMSs, but these do not scale to the Internet environment. In this thesis we propose an incremental and dynamic method for replicating and reallocating fragments of database relations in a highly scalable manner. Our algorithm can execute data redistribution without requiring prior knowledge of the global environment, does not require redistribution from scratch when the environment changes, considers the relationship between relations (fragments), and reacts dynamically to changing access patterns. The algorithm simulates a "spring system" among the fragment and query objects, where correlations (constraints) between objects play the role of springs. In a real spring system, the springs pull the objects in such a way as to approach a stable equilibrium state; similarly, the constraints cause the fragments and query objects to be relocated until a stable equilibrium is reached. We show that the locations of the objects at equilibrium correspond to the optimal arrangement for efficient allocation.

Acknowledgements

Studying at the University of Waterloo was an enriching experience - for having the opportunity to upgrade my knowledge, for sharing ideas with people with same interests, for all the professional challenges.

I owe all of these to my supervisor, M. Tamer Özsu who gave me the opportunity of joining the Masters program. Many thanks for his constant guidance and support, and for the time he took to advise me whenever I was in doubt about the research topics. I am grateful for his assistance through the whole process of researching and preparing this thesis. His vast knowledge and inspiring ideas helped me overcome many obstacles, stay focused, and gain confidence in my skills. Having him as supervisor shaped my thinking for the years to come.

Special thanks to Ric Holt for his financial support during the last two terms of my graduate studies and to Ken Salem and Steve MacDonald for reading the thesis.

Last but not least, I would like to thank my colleague Ning Zhang for helping me in various situations, and also to James She and David DeHaan for being great project partners for the courses I took at Waterloo.

Contents

1 Introduction **1**
- 1.1 Why is the Allocation Problem Important? . . . 1
- 1.2 Brief Overview of DDBMS . . . 2
- 1.3 Problem Scenario . . . 2
- 1.4 Problem Parameters . . . 4
- 1.5 Why is the Allocation Problem Hard? . . . 5
- 1.6 Thesis Organization . . . 7

2 Related Work **8**
- 2.1 Fragmentation . . . 8
  - 2.1.1 Vertical Fragmentation . . . 9
  - 2.1.2 Horizontal Fragmentation . . . 9
  - 2.1.3 Mixed Fragmentation . . . 9
- 2.2 Replication . . . 10
  - 2.2.1 Benefits of Replication . . . 10
  - 2.2.2 Replication Protocols . . . 11
- 2.3 Data Allocation . . . 12
  - 2.3.1 Static Allocation Approaches . . . 13
  - 2.3.2 Dynamic Allocation Problem . . . 14

3 The Spring Algorithm **22**
- 3.1 Environment Description . . . 22
  - 3.1.1 Sites . . . 22
  - 3.1.2 Data fragments . . . 23
  - 3.1.3 Network Links . . . 23
  - 3.1.4 Data Access Framework . . . 23
- 3.2 Cost Constraints . . . 24
  - 3.2.1 Query Access Constraints . . . 25

3.2.2 Correlation Constraints . . . 27
3.2.3 Replication Constraints . . . 27
3.3 The Spring System Approach - An Analogy . . . 28
3.3.1 The Objects . . . 28
3.3.2 The Springs . . . 28
3.3.3 The Sites and The Network Links . . . 30
3.3.4 System Equilibrium . . . 30
3.4 The Spring Algorithm . . . 32
3.4.1 Graph and Placement . . . 32
3.4.2 Bubble Definition . . . 33
3.4.3 The Bubble Energy . . . 34
3.4.4 Replication, Move, and Delete Techniques . . . 37
3.4.5 The Spring Algorithm - Description . . . 39

4 Experiments **45**
4.1 Experimental Setup . . . 45
4.1.1 Data Generator Module . . . 47
4.1.2 Data Cleaning Module . . . 50
4.1.3 Spring Algorithm Simulator Module . . . 51
4.1.4 Experimentation Platform . . . 52
4.2 Experiments and Results . . . 54
4.2.1 Validation Experiments . . . 54
4.2.2 Performance Evaluation Experiments . . . 61
4.2.3 Conclusions . . . 70

5 Conclusions and Future Work **72**
5.1 Overview . . . 72
5.2 Contributions . . . 75
5.3 Future Work . . . 76

Bibliography **79**

List of Tables

2.1 Propagation (Eager or Lazy) vs. Ownership (Master of Group) 11
2.2 Dynamic Allocation Algorithms 14

4.1 The Default Values in the Experiments 53
4.2 The Validation Experiments 55
4.3 Performance Evaluation w.r.t. Nr. of Sites 63
4.4 Performance Evaluation w.r.t. Nr. of Fragments 64
4.5 Performance Evaluation w.r.t. Nr. of Queries 65
4.6 Performance Evaluation w.r.t. to S/J, U/R, and Query Access Distrib. . . 69

List of Figures

1.1 Servers' P2P Architecture . . . 3
1.2 Problem Parameters . . . 4

3.1 Cost constraints . . . 26
3.2 Analogy between the distributed database system and the spring system . 29
3.3 The two layers of the problem . . . 30
3.4 Object connected with springs . . . 31
3.5 $\vec{f_r}$ should be greater than the $\vec{f_f}$ to move the object. . . . 31
3.6 A reallocation affects more the closer replica objects. . . . 33
3.7 All fragments are non-replicated and uncorrelated . . . 34
3.8 Non-replicated but correlated fragments. . . . 35
3.9 Replicated but non-correlated fragments . . . 36
3.10 Replicated and correlated fragments . . . 37
3.11 Replication step in the Spring System . . . 38
3.12 After $R_2(F)$ is moved, Q_2 uses $R_1(F)$. . . . 39

4.1 Module interconnection . . . 46
4.2 The Spring Algorithm Simulator . . . 51
4.3 The heuristic rearranging algorithm analyzes one fragment at a time. . . . 56
4.4 Replica objects' order of analysis is important. . . . 57
4.5 In a small setting it is crucial to reconnect the springs for each simulation. 58
4.6 In large environments spring reconnection is not important. . . . 59
4.7 The number of analyzed sites is important. . . . 60
4.8 Randomly generated sites encourage replication. . . . 62

List of Algorithms

1 Finding the best allocation 6
2 GetBubbleEnergy 42
3 CheckMove 43
4 CheckReplicate 43
5 SpringAlgorithm 44
6 Rearrange the springs 50

Chapter 1

Introduction

1.1 Why is the Allocation Problem Important?

With the current popularity of Internet and e-business, distributed database systems (DDBMS) are widely deployed to provide the back-end support for Web-based database applications. Efficient data distribution (allocation) is critical to ensuring cost-effective performance and high availability of data for applications or services [46]. The allocation problem is important since a good data location, close to the users accessing it, can boost the DDBMS performance and reduce access costs.

In traditional DDBMS, the allocation scheme is determined at distributed database design time, and it usually remains fixed. However, the optimal data allocation depends on factors that change dynamically at run-time (read/update ratio, network connectivity), and on the optimization goal (e.g. cost, performance, availability). The various replication and allocation strategies proposed for data distribution in traditional Distributed DBMSs are not suitable for a system running in the Internet environment where volatility requires adaptivity. These strategies are not able to react easily to changes in access patterns (changes in query workload or user distribution). Also, most of them do not consider the correlations between data fragments[1], which represents a significant factor that has an impact on how and where data should be moved or replicated.

In this thesis, our goal is to *find a way to replicate and dynamically allocate the database fragments to various sites in order to run the queries faster.* Since we deal with a dynamic environment, a static approach to the allocation problem would fail from the start. Therefore, we propose an incremental and dynamic algorithm which can redistribute data without knowledge of the global environment in advance, that does not run from scratch, and that

[1]We are not looking at parallel DBMSs.

reacts dynamically to changing access patterns. If the fragments have affinities between them, not only the fragment that changed its access pattern should be redistributed, but also others that could be influenced by the allocation. Our algorithm takes as input the current allocation configuration and access patterns, and, based on these, it computes new locations. For reasons which are detailed later, we call this algorithm the *Spring Algorithm*.

In this thesis we do not try to solve the fragmentation problem, although it is directly related to allocation (changing the database structure directly affects the way allocation is performed). Our algorithm assumes that fragmentation has already been completed, and does not try to re-fragment.

1.2 Brief Overview of DDBMS

This overview aims at giving a clear picture of the environment within which our problem is formulated.

"A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is defined as the software system that permits the management of the DDBS and makes the distribution transparent to the users." [41]

Two of the commonly used distributed database architectures are the client/server and peer-to-peer systems:

- Client/Server Systems: Generally speaking, a client can only make requests to a server site, which executes the queries and returns the results. The server can also be the client of another server. More details about client/server systems can be found in [41] and [18].

- Peer-to-Peer Systems: These are fully distributed architectures, in which each node of the system can work as a server that stores part of the database and as a client that runs queries. Details about P2P can be found in [6, 30, 37, 7, 38, 23, 16]. A P2P and databases bibliography can be found at www.cs.toronto.edu/db/p2p/.

1.3 Problem Scenario

The system we consider consists of a large number of sites, each of which has full database functionalities (Figure 1.1). The users run queries on these sites. For our purposes, the way the users connect to sites is transparent, and the costs are not considered. A user

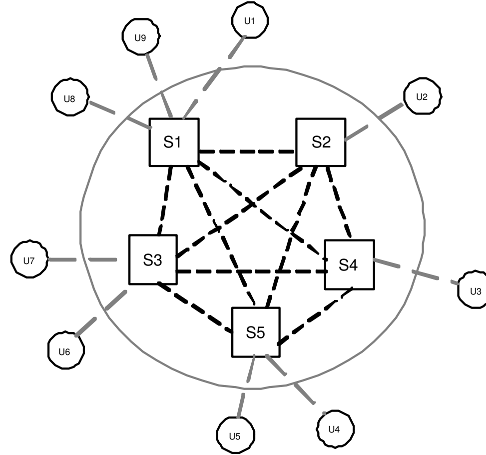

Figure 1.1: Servers' P2P Architecture

accessing some fragments should not be aware if the data are replicated or not, or where the data are located.

The data can be fragmented and replicated, and the unit of distribution is a fragment[2]. The sites cooperate to reduce the overall cost of accessing data. The change of frequency when accessing those fragments triggers their movement, replication, or deletion.

We assume an initial distribution of fragments. When the access pattern to a fragment (or replica) changes, the system reconfigures itself dynamically. The queries accessing the fragments represent the factors that cause the data reallocation, as the goal is to minimize the cost to execute them.

As the system is placed in the Internet environment, there are few characteristics associated with it:

- Scalability is the main issue. The system must be highly scalable in order to perform well in an environment where the only constant is change, and users access the sites at a high rate.
- The user population and the user access patterns change from time to time at each site, since the users' interests vary constantly.

[2]The fragment is defined as a part of a database relation.

1.4 Problem Parameters

Figure 1.2 describes the three types of problem parameters: sites, queries, and database capabilities.

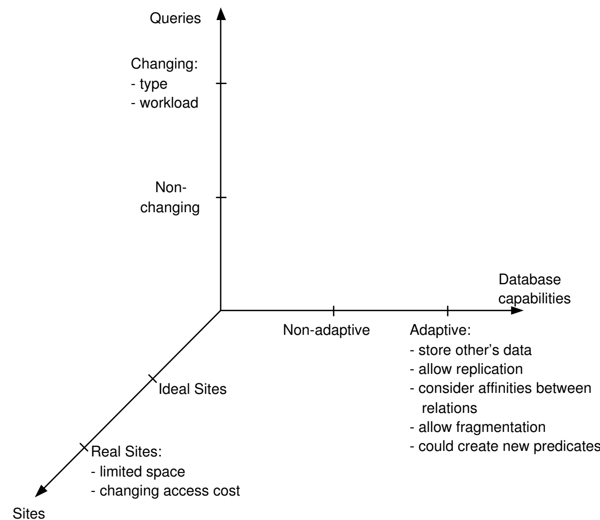

Figure 1.2: Problem Parameters

Ideal sites have an infinite storage space and always have the same access cost (including the network transmission time). This means that delays caused by overload never happen and the network is never a delay factor. *Real sites* have limited space, due to the secondary storage (i.e. hard disk) limitations. If more space is needed, changes have to be done to the site. Also, the access cost to a site might change due to network or server overload. As opposed to the static approaches, the algorithm we propose (Spring Algorithm) takes into account that, over time, the access cost to a site can change due to a network failure or overload, and the sites have limited space.

The *queries* accessing the DDBMS may or may not change over time. The Spring Algorithm deals with changing queries as well.

The *database systems* are of two types. A *non-adaptive* system is only capable of providing data to the users, and is not capable of adapting to the changing conditions. The *adaptive* system, on the other hand adapts to environmental changes to reduce the global access cost. It is capable of storing data that may be moved to it from other sites and allows replication if this represents an improvement in the global cost.

Each point in the three-dimensional space (Figure 1.2) is a version of the allocation problem, which increases in complexity from the simplest case (ideal sites, non-changing

queries, and non-adaptive databases) to the most complex case (real sites, changing queries, and adaptive databases). The Spring Algorithm works with the most complex case.

1.5 Why is the Allocation Problem Hard?

The general problem of data allocation is an $\mathcal{NP} - hard$ problem since *each site has limited space available* to store the fragments [41, 19]. We use Algorithm 1 to illustrate the difficulties in finding the best allocation. Let us consider a set of f fragments, s sites and q queries. In this algorithm there are three sub-tasks:

1. **Finding how many copies of each fragment are to be allocated in the best distribution.** For $f = 3$ and $s = 4$, assuming that all the fragments allow replication, here are some example of possible allocation situations: (1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4), ... , (4, 4, 2), (4, 4, 3), (4, 4, 4). The pair (i_1, i_2, i_3) represents the number of replicas each of the three fragments have. The last pair shows that each fragment has four copies which will obviously be located on each of the four sites. There is no advantage in having two copies of a fragment on one site, therefore the maximum number of replicas a fragment can have equals the number of sites. Consequently, there are $f \times s$ possible combinations. All of these are captured in line 2 of Algorithm 1.

2. There are $f \times s$ combinations of numbers of replicas of each fragment to be allocated. If the number of fragments and sites increases, the number of combinations "explodes". But that's not all. For each of the above pairs there are many possible locations **where the replicas can be allocated** (sites 1 to s). Using the same values of f and s as in the above bullet, let's consider the pair (2, 2, 3). The first fragment has two replicas that can be allocated to the following sites: {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}. This is similar for the second fragment. The third fragment's replicas can be allocated as follows: {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}. Only for this subproblem there are $6 \times 6 \times 4 = 144$ cases to analyze. To generalize, if there are n_1 replicas of the first fragment, n_2 replicas of the second fragment and so on until n_f, then there are

$$\prod_{i=1}^{f} \binom{s}{n_i}$$

possibilities where the replicas can be allocated. All of these are captured in line 5 of Algorithm 1.

3. After the two previous steps, a large set of fragment allocation configurations is obtained. Now, for each configuration, **finding whether this is the best allocation** requires checking if the overall cost of executing the queries is minimum. Regarding the queries, it is known which fragment each query is accessing. However, which replica of an accessed fragment should be used when running a query is still unknown. For instance, for an allocation configuration such as { {1, 2}, {2, 3}, {1, 3, 4} }, a join between two fragments $J = F_2 \bowtie F_3$ can be executed using any of the replicas of F_2 located at sites 2 or 3, and any of the replicas of F_3 located at sites 1, 3 or 4. Therefore, for this query there are six possibilities that have to be tried. To generalize, for a query accessing p fragments, each having r_i replicas, there are $r_1 \times r_2 \dots r_p$ possible ways to run the query. All of these are captured in lines 6, 9 of Algorithm 1.

Algorithm 1 Finding the best allocation

```
1: {Get all sets of possible number of replicas of each fragment}
2: for all sets { (i_1, i_2, ... i_n) }, numbers of replicas of each fragment do
3:    {Get the set of set of sites where the replicated fragments are placed}
4:    { {S_{1,1}, ...S_{1,i_1}} are the sites on which the i_1 replicas of first fragment are placed}
5:    for all sets { {S_{1,1}, ...S_{1,i_1}} ... {S_{n,1}, ...S_{n,i_n}} } of different sites do
6:       for all q query accessing fragments {F_{k_1}, ... F_{k_m}} do
7:          { The goal is to use the closest copies of the fragments to serve the query }
8:          { Assuming F_{k_1} has a replica stored on S_{k_1,p_1}, F_{k_2} has a replica stored on S_{k_2,p_2}
            and so on F_{k_m} has a replica stored on S_{k_m,p_m} as described above }
9:          for all {S_{k_1,p_1}, ...S_{k_m,p_m}} s.t. S_{k_i,p_i} ∈ {S_{k_i,1}, ...S_{k_i,i_{p_i}}} do
10:            If is the best configuration, save it
11:         end for
12:      end for
13:   end for
14: end for
15: Return the saved configuration
```

As shown in the above example, the number of cases that need to be analyzed increases exponentially with the number of sites, fragments, and queries. The running time to place f fragments on s sites is $f \times s$. However, the limited space on the sites could lead to removal of fragments to make room for others if this results in a better configuration. Therefore, the search for the best allocation becomes difficult, making the allocation problem $\mathcal{NP}$-hard.

1.6 Thesis Organization

This chapter has set the general framework of the allocation problem and outlined the strategy we propose. The rest of the thesis is organized as follows:

- Chapter 2, *Related Work*, is an overview of previous allocation strategies, briefly touching two close related strategies: fragmentation and replication.

- Chapter 3, *The Spring Algorithm*, focuses on the spring system approach and describes the Spring Algorithm. This chapter also contains a description of the environment in which our algorithm operates.

- Chapter 4, *Experiments*, presents the experimental results of the simulation model that was implemented to analyze the performance of the Spring Algorithm. This chapter describes the experimentation platform and the two classes of experiments: validation and performance evaluation.

- Chapter 5, *Conclusions and Future Work*, gives an overview of the proposed strategy and experimental results, presents our contribution, and outlines potential future work.

Chapter 2

Related Work

This chapter presents an overview of previous dynamic distribution strategies, which are of interest in this thesis, and briefly introduces static distribution strategies as well. We also present two orthogonal strategies to the general problem of data distribution – data fragmentation and replication – as they all combine to improve the efficiency of data distribution.

Every distributed database system is described by a *global conceptual schema*, which specifies the logical structure and information of all the data contained in the distributed database. In the case of distributed relational database systems, the global schema contains global relations. The relations are divided into smaller logical data units called *fragments*, which helps to localize accesses and reduces the amount of irrelevant data accessed for any transaction. The mapping between the global relations and fragments is defined in the *fragmentation schema*. Similarly, there is an *allocation schema* which specifies the location of each fragment (or each replica of a fragment) according to the distribution algorithm used.

The chapter is organized as follows: Section 2.1 provides a short summary of the fragmentation strategies, Section 2.2 gives an overview of the replication strategies, and Section 2.3 details allocation strategies that improve the transfer costs.

2.1 Fragmentation

Fragmentation is the process of partitioning a relation into sub-relations. It is preferable that the parts of the relation that are accessed together to be located in the same fragment. The benefits of fragmentation are the following:

- **Locality:** Many queries involve only a subset of a relation. This subset of data is

captured in a fragment that, ideally, should be stored in a place close to the queries accessing it. The decreased space and increased locality effect can dramatically improve the access performance.

- **Concurrency:** The query processing is sped up when a query is translated into subqueries and applied to multiple fragments in parallel. This allows concurrent processing of data and distributed accesses at multiple sites for higher system throughput [39].

A fragment's *granularity* should be decided in a reasonable manner. It can range from several tuples or attributes to full relations. If the granularity is large, it will consume excessive bandwidth for data delivery. However, if the fragments are small, this will degrade the performance due to extra interactions and processing required to merge the fragments.

2.1.1 Vertical Fragmentation

In vertical fragmentation a relation is split vertically into a set of fragments by projecting a relation over a subset of its attributes. Each fragment will contain a subset of attributes as well as the primary key of the relation. The primary key is used to reconstruct the fragments, by joining the fragments. More details about vertical fragmentation can be found in [41].

2.1.2 Horizontal Fragmentation

In horizontal fragmentation, which is by far the most commonly used fragmentation scheme, a relation is divided into a set of fragments by rows. Each fragment contains a subset of tuples of the original relation. Horizontal fragmentation is proven to improve the performance of database systems [5]. It is used for three reasons: it reduces the disk access required to execute an application, by minimizing the number of irrelevant objects accessed; it reduces data transfer among sites; and it allows the application to be executed concurrently for a higher degree of parallelism. In relational systems, the horizontal fragmentation is achieved by selecting a set of tuples that satisfy a condition. More details about vertical fragmentation can be found in [41].

2.1.3 Mixed Fragmentation

Sometimes a single strategy of horizontal or vertical fragmentation may not be optimal. Partitioning horizontally and then vertically or vice-versa may achieve better results. It can even be done simultaneously.

2.2 Replication

This section reviews different replication strategies. First we introduce the advantages of replication and then we briefly discuss several characteristics of replication protocols.

2.2.1 Benefits of Replication

In DBMSs replication is generally used due to three important benefits:

- High availability
- Fault tolerance and
- Performance enhancement.

For any critical system, one might want to have data replicated for **high availability**. If server failures occur, a user should be able to get the data from another server. This is an important area of interest for researchers due to the increasing popularity of mobile computing devices. Distributed systems textbooks [18, 41, 45] talk about these issues in detail.

Fault tolerance refers to resilience of a system to failures [32, 13]. It is important to implement a method to check the correctness of data when this happens, as sites that are not working properly can have adverse effects on others. A properly-running server A that gets an update may try to propagate it to a site B that failed. A will be forced to wait until the time-out period expires to realize that B is down, resulting in delays of an operational site.

Many papers talk about obtaining **performance enhancement** in a Web environment by locating replicas on sites close to the users accessing them [28, 24, 9]. Content Delivery Network technologies like Akamai [1], Mirror-Image [2], and SinoCDN [3] applied this strategy with success. This strategy is similar to caching because they both place copies close to the users. However, there are several subtle differences such as the way the copies are removed and deployed, the update protocol, the impact on the catalog, and the granularity. All these differences are explained in detail in [35]. Replication allows load balancing as user requests are distributed to many servers, reducing the chances of server bottlenecks. It also reduces the network transfer and the inherent latency to which the users are exposed because data are always retrieved from the closest location. However, replication has a side effect that needs to be considered – the cost to maintain the consistency of copies. Since maintaining data consistency is not a trivial task, many algorithms have been proposed to efficiently keep data consistent; they are called *replication protocols*.

2.2.2 Replication Protocols

Different applications have different requirements about the quality (freshness etc) of data they access. A replication strategy that works well with one class of applications may not work well with others. Many replication protocols have been proposed [20, 4, 51, 10, 40] to satisfy these requirements with minimum consistency costs.

Table 2.1 presents a taxonomy of the replication protocols based on object ownership strategies (master or group) and propagation protocols (eager and lazy) [22].

Table 2.1: Propagation (Eager or Lazy) vs. Ownership (Master of Group)

Propagation versus Ownership	Eager	Lazy
Master	N transactions, one object owner	One transaction, one object owner
Group	N transactions, N object owners	One transaction, N object owners

In **master replication** (also called passive replication), one replica is considered the source (primary copy). The updates are applied to the primary copy, which then forwards them to the rest of the replicas (that are read only). Whenever the primary copy fails, another replica is selected to take over the role of primary copy.

In **group replication** (also called state machine approach or active replication), each replica has equivalent roles. The users send updates to any replica and the updates are propagated to all the copies.

The **eager schemes** ensure that changes to replicated data are performed within a transaction boundary. The transactions commit on all sites that have the same data and all of these replicas have to be updated before the transaction commits. This is a good approach for applications that cannot afford stale data.

The **lazy replication** protocols, by contrast, focus on performance and therefore allow some degree of inconsistency. Since it is generally considered that most applications allow a certain level of inconsistency, many papers deal with this type of replication. Two strategies, shadow copies [8] and bounded errors [51, 40], are commonly used to reduce the message overhead in lazy replication. In lazy replication protocols, the level of consistency can be determined by the precision of data or the freshness: (1) How precise is the information provided?; (2) How fresh are the replicated data? The value cannot be older than some threshold, but there is no specified information about the deviation from the real value.

2.3 Data Allocation

Data allocation deals with finding the best (or close to the best) locations to place fragments in various environments such that specified constraints are satisfied. There are strategies that try to minimize the storage space [34, 29], the bandwidth [47], the number of replicas deployed [36, 42, 14], and/or the total communication cost [26, 15]. The constraints may be defined on the average number of nodes that users have to pass to get to the closest object they access, on the maximum wait-time of each client, etc.

There are three types of allocation alternatives: non-replicated and fully replicated allocations, which are the two extremes, and partially replicated allocation.

Non-Replicated Allocation:

In this alternative, the fragments are not replicated. If they are updated frequently or have a low read frequency, or if there is a security issue, it is better to have only one copy. However, a non-replicated fragment is vulnerable if the site hosting it fails. Also, the site might be a performance bottleneck.

Fully Replicated Allocation:

In this case, each fragment is located at every site, therefore the allocation problem is trivial. This is feasible only if the applications are critical and if either the data requests are generally read-only or the updates can be performed off-line and in batches. A catalog that keeps track of the fragment's location is not required. The limitations of this type of allocation are: (1) utilization of the storage space is poor when replicas are rarely accessed, and the storage cost may be high; (2) the cost to maintain consistency of a large number of replicas can be significant since, for each update message, all sites must communicate; and (3) as presented earlier when discussing *fault tolerance*, in replication algorithms like those enforcing strict consistency, whenever a site fails all other sites must wait for the failed site to recover.

Partially Replicated Allocation:

Since both non-replicated and fully-replicated allocation have limited utilization, it is natural to consider partial replication. In this case, the number of replicas of each fragment varies based on the access patterns. This is the common approach used in practice, as spatial and temporal locality can be fully leveraged. However, for partial-replication two issues must be addressed: (1) *How many copies of a fragment should be replicated in order to get the best performance?* This is a complex task. On the one hand, the number of updates, the cost of updates, and the storage capacity of the sites limit the number of replicas to be deployed. On the other hand, frequent reads in clustered geographical regions that result in high gains from reading data locally motivate an increase of the number of replicas. Considering these conditions simultaneously, an optimal or close to optimal number of replicas is obtained [26, 50]. (2) *How should the fragments be allocated to*

the sites? There are many papers analyzing data allocation. Some of them propose static allocation techniques, assuming that access patterns to data and network characteristics do not change. Others analyze the allocation problem in a dynamic environment where replicas have to be reallocated to adapt to changing conditions. We discuss this issue in detail in the following sections.

In Internet environment, non-replicated and fully replicated techniques are not useful due to the high read frequency and the large number of sites respectively. Also, static allocations are not appropriate in dynamic environments, therefore, in this thesis we focus on dynamic allocation strategies.

2.3.1 Static Allocation Approaches

Traditionally, database allocation problems have been performed by off-line analysis and optimization. In the static allocation strategies, the access patterns are known beforehand and the fragments are located permanently. This approach works well for static access patterns, but in the Internet environment, which is highly dynamic, the static allocation strategies are not adequate. Consequently, they are only briefly presented in this section.

In [42] M replicas are located on N nodes of a Content Distribution Network, and the goal is to find the optimal location in order to reduce the cost to access the data replicated on the servers. In [29], the goal is to minimize the number of hops a client has to go through to get the data while satisfying the storage space constraints, whereas [26] proposes two algorithms to minimize the total communication cost, without any constraints on the sites' space. Something new in data allocation is introduced in [33]. The authors propose a new parameter, *confidentiality*, to be the third deciding factor on data allocation, besides the commonly used read/write frequency. If data are replicated on many sites, the confidentiality decreases. Some data should have a certain level of confidentiality, therefore it will have a bound on the number of replicas. The optimal location of replicas in a network that is using Read-One-Write-All replica consistency policy is considered in [15]. The authors show that, for a path graph with integer distances and the cost between the nodes given by the length of the shortest path, the allocation problem can be solved in polynomial time.

More allocation strategies are presented in [12] (network cost constraints), [34] (storage space constraints), [36] (replication constraints). A comparison among different static allocation algorithms can be found in [31]. We will not consider static allocation approaches any further, as we are interested in techniques that can dynamically adapt to new network and system conditions.

2.3.2 Dynamic Allocation Problem

As mentioned, static placement of data has limitations because the allocation does not react to changing conditions. An ideal solution is to incrementally change the allocation schema according to new information towards the "best" allocation, requiring periodic relocation of the fragments. Fragments that are highly read or updated should be moved as fast as possible because, during the time they are moved, the system's execution is suboptimal [25]. It is not feasible to stop the access service until data migration has been completed. If the size of the fragment is significant, the duration of the interruption is long. [48] deals with this problem by using message forwarding. As in DDBMSs there are correlations between fragments, a good strategy should also consider correlation.

Because the problem of fragment allocation is $\mathcal{NP}$-hard, heuristic algorithms have been proposed to get a "close to the best" allocation. We categorize them in Table 2.2, and we refer them by the papers in which they were presented.

Table 2.2: Dynamic Allocation Algorithms

Type of Algorithm vs. Type of Network	Distribution Algorithms	Caching Algorithms
Tree-Shaped Networks	[49] [50]	[14]
General Topology Networks	[17], [11], [27], [43], [44] [50]	[47]

We have found several dynamic distribution algorithms [49, 50, 17, 11, 27, 43, 44] and also two caching algorithms [14, 47] that, as a side effect, move and replicate data, and therefore are of interest. Based on the topology of the network on which algorithms are used, we have found that three of them only work in tree-shaped networks (tree-like communication control) [49, 50, 14]. Notice that [50] is also placed in the general topology network because the authors also propose a variation on the original algorithm that is adapted to this type of network.

Distribution Algorithms in Tree-Shaped Networks

The practical Dynamic-Data-Allocation (DDA) algorithm is one of the early proposals for dynamic allocation of replicated data [49]. This algorithm changes dynamically the replication scheme of an object, as objects' read/write pattern changes in the network. The goal is to optimize the total communication cost to access an object, based on its

current read/write pattern. Each object in the database has a *primary copy*. The site that doesn't store the object it accesses reads and updates the primary copy. If the site stores a copy, the reads are executed locally and the updates are propagated to the primary copy. The primary copy will then forward the updates to the other copies. As long as the primary copy is located on an up-and-running site, the failure of a site storing a copy does not affect other sites storing another copy. If the primary copy fails, an election protocol is executed to select another primary.

The cost to read an object locally is 1 (processing cost), while the cost to read it from a remote site is $1 + d$, where d is the cost to retrieve the object. Considering there are c copies, the cost for a write is $c \cdot (1 + d) - d \simeq c \cdot (1 + d)$. Each site i performs $\#W_i$ writes and $\#R_i$ reads in a *time unit*. Initially, the algorithm allocates copies of the objects on each site where there is a read. Next, it considers the updates and, if the benefit from reading data at each of those sites is less than the cost to maintain its consistency, the copy is discarded. After the allocation is performed, the sites k that keep a copy are $\{k|(1 + d) \cdot \sum_i \#W_i \leq d \cdot \#R_k\}$. A strategy to maintain a specified level of availability is also proposed. If there is a constraint (i.e. "the number of copies cannot decrease below a threshold, say t") the primary copy refuses to drop a copy from a site even though the update cost is higher than the read cost. The inefficient copy is removed only if a new copy is allocated.

The approach is interesting and the algorithm is simple to implement, but there are three drawbacks. First, the cost to move an object from one site to another is not considered, and this cost may not be negligible. The second drawback is that, in this approach, whenever a site gets either a read or a write operation, it runs a test to check if the copy it stores is suitable for this location. If the reads and the updates are interlaced, it could happen that a copy is dropped after each write message and brought back immediately when a local read is issued. The most important drawback, which makes this algorithm inapplicable to an Internet-based DDBMS, is that the algorithm does not consider correlation between fragments.

The other distribution algorithm that works in tree-shaped networks is the Adaptive Data Replication (ADR) algorithm [50]. It is proposed by the same authors and has the same goal of reducing the communication cost of accessing an object by dynamically changing its replication scheme. Initially, there is only one copy of each object, and the objects are replicated and/or moved according to their read/write pattern. An object is allocated at n sites and a copy is found at every site along the path between any two sites storing copies of that object. This means that a copy is allocated in all nodes of a subtree of the tree shaped network, therefore each object has its own allocation subtree. The reason behind this strategy is that, when the updates are propagated to the leaves

of the allocation subtree, the copies in between can be updated at no cost. The costs are computed from the number of local reads minus the total number of writes, just like in [49]. Periodically, at the end of a fixed time interval, each site that is a leaf of the allocation tree for a set of objects performs tests on each of the objects in that set. These tests are expansion, contraction, and switch. In the *expansion test*, a site x replicates an object at a neighbor site y if the number of read requests coming from y in a given time interval is greater than the total number of updates received by x from any site other than y. If the test fails, the *contraction test* is conducted to verify whether or not the object on x should be removed. Since x is a leaf of the allocation subtree, it has only one neighbor y that stores the object. If the number of writes received from y in the last time interval is greater than the number of reads x receives, the object is dropped. The *switch test* is performed only if the expansion test fails and the object is stored at only one site (singleton). For each neighbor node of the site storing the singleton object, the site counts the number of accesses to the object in the last time interval. If there is a node x such that the number of requests coming from x is greater than the sum of all requests from all other nodes, the object is moved to x.

This algorithm can also be used in general topology networks, as the minimum spanning tree can be computed and then the algorithm can run on it. However, there are disadvantages since minimum spanning tree does not guarantee the shortest distance between two nodes. ADR-G is a version of ADR which uses the shortest path for a general topology network. Another issue is that, since the storage space is limited, it is not feasible to have a large number of sites storing the same data, especially in a large environment like the one we try to address. Also, correlation is a must but is not addressed here.

Distribution Algorithms in General Topology Networks

An original approach to the problem of distributed database allocation is presented in [17], where a genetic algorithm[1] is used to address this problem. In distributed database allocation, a set of n fragments is allocated at m sites, where each fragment j is characterized by its size s_j and each site i has its own capacity c_i. Each fragment is required by at least one of the sites. The sites' requirement for each fragment is defined in an $m \times n$ requirement matrix, where $r_{ij} \geq 0$ indicates the requirement of site i for fragment j. Also, the transmission cost is given by an $m \times m$ transmission cost matrix, where t_{ij} indicates the cost of site i to access a fragment located on site j. The goal is to find a placement $P = \{p_1, p_2, ..., p_n\}$, where $p_j = i$ indicates that fragment j is located at site i, provided

[1]The genetic algorithms [21] are adaptive search techniques based on the principles and mechanisms of natural selection and "survival of the fittest" from natural evolution.

the capacity of any site is not exceeded,

$$\sum_{j=1}^{n} r_{ij} \cdot s_j < c_i \qquad \forall i | 1 \leq i \leq m$$

and the total transmission cost,

$$\sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} \cdot t_{ip_j}$$

is minimized.

In genetics, each chromosome is formed of genes, and the genes' values are called alleles. In the proposed algorithm, the genes represent the fragments, and the alleles represent the sites on which the fragments are located. Each chromosome can be represented as a string of integer values and is an encoding of a placement P in the distributed database allocation problem. At each iteration, a portion of the population of chromosomes is selected and somehow altered, then reintroduced into the population pool. Chromosomes are probabilistically selected for reproduction based on the "survival of the fittest" principle. The fittest chromosome is the one that has a low cost function (defined above). The offspring are generated through a process called crossover [21], which can be improved by mutation [21], and then are introduced into the pool. Using this process, chromosomes corresponding to good allocations are the survivors in the population pool. The performance experiments show that the genetic algorithm gives a better allocation than a greedy algorithm in which each fragment is placed in turn in the least cost location.

The approach is interesting, but it fails to consider replicated fragments. Also, the algorithm cannot be applied in large environments as the computation to maintain the chromosome population is high. Also, the site capacity is not the most relevant constraint in the Internet environment, where bandwidth is the parameter that affects the allocation the most.

Two dynamic algorithms, *Simple Counter* and *Load Sensitive Counter*, are proposed in [11]. The goal is to dynamically reallocate data in a partitioned distributed database, just like in [17]. In the first algorithm, each fragment keeps a counter for each site. Whenever the fragment is accessed from site S_i, the ith counter is increased. To discount prior samples, the algorithm uses an *aging factor* that ascribes high values to new entries in the counter and low values to old ones. Every time interval, t_{check}, the fragment is moved to the site with the highest number of accesses. The value of t_{check} should be low enough for the system to respond quickly, but large enough to prevent premature signaling of change in access patterns. The second algorithm considers load balancing, which is shown to be an

important practical consideration when reallocating data. The system load and the access frequencies are monitored. A fragment is not moved to the site with the highest number of accesses if that site would be overloaded. There are three reasons why the algorithms would perform poorly if used in the Internet environment. First, the algorithms do not consider locality. If a site from Halifax accesses a fragment 101 times, and three sites in Vancouver, Victoria, and Edmonton access it 100 times each, the fragment is located in Halifax. Second, both replication and fragment correlation are a must but are not considered. Third, just counting the queries that access a fragment is not sufficient, since in the real world query selectivity is also important. This simplification could only be used if the fragment is fully accessed, or if all queries access the same size of the fragment.

Mariposa [27] is a distributed database system based on an economic model. However, it is relevant to this thesis because it also finds the best allocations for fragments. The queries are executed by organizing auctions. When a client issues a query, a budget is allocated for that execution. The budget shows the importance of the query, creating a way to deal with priorities. A broker creates an execution order for each query operator. Then it organizes an auction where all sites that contain full or partial data or are willing to execute some query operators are bidding and offering their services. The broker chooses the execution plan with the lowest cost (the paper interprets this execution plan as a broker trying to maximize its own profit). The faster the query operators are executed, the more money a site makes. A site S_2 can buy a fragment from S_1 for a period of time t in which S_1 will forward the updates. S_2 pays S_1 at least for the effort to propagate the updates during t, and for the loss caused by the fact that some queries are now executed at S_2 instead of S_1. S_2 recuperates the money spent by running the queries faster, therefore making more money. None of the papers talking about Mariposa present how efficient the data redistribution is.

A migration and replication protocol for Internet hosting is presented in [43]. The goal is to place replicas of Internet objects in the vicinity of a majority of requests, while ensuring that no server is overloaded. The protocol relies on information available from routing databases and IP headers.

There are two pairs of tunable parameters in the protocol: low and high watermark for the hosts (lw and hw), and deletion and replication threshold for the objects (u and m). The watermarks add stability to the system, and reflect the system's capacity. If the load at a site exceeds hw, the performance degrades. The deletion and replication thresholds determine the replica placement. The algorithm keeps an access count for each object, similar to [11]. If the deletion and replication count of a replica drops below the deletion threshold, the replica is removed. For a count between the two thresholds the replica

can only migrate. Another copy is created only for counts greater than the replication threshold. To provide stability, the constraint $4u < m$ must be satisfied. Intuitively, this should be enforced such, that after replication, every replica has an access count exceeding u, therefore no newly created replica is deleted. This avoids cycles of creations and deletions of replicas. If a site's load exceeds hw, it switches to off-loading mode and moves objects to other hosts even though the performance degrades due to proximity issues. The process stops when the low watermark, lw, for hosts is reached. A candidate accepts a replication request if the load is below lw. To accept a migration request, a candidate site, in addition, checks if the upper bound load would be below hw. This prevents situations in which a single object migration would bring the recipient load from below the lw to above hw. The replication is not so strictly conditioned, as the load when accessing the replicated object will be shared by the two sites. The strategy proposed by [43] is based on server workload only, because they deal with an Internet hosting service which is highly accessed. However, in what we try to address, the server load is important but the transmission cost is the primary metric. Also, the authors do not consider correlations between Internet objects.

A large peer-to-peer file-sharing system is described in [44]. The goal is to achieve high availability by replicating files automatically and in a decentralized fashion. The minimum number of file replicas required to satisfy the availability threshold is calculated dynamically. Each of the sites has the same probability p to be up and the replica location service has a specified accuracy RL_{acc}. The cost function represents the availability of the files (computed as the probability that at least one replica is available times the accuracy of the replica location service). The number r of required file replicas is obtained from $RL_{acc} \cdot (1 - (1 - p)^r) \geq Avail$, where $Avail$ is the imposed level of availability. Once a site knows the optimal number of replicas for a file it stores, it employs the replica location service to discover how many replicas actually exist. If the number returned, say M, is less than r, $(r - M)$ copies of the file are created and distributed to remote locations.

The authors assume that the resource discovery mechanism provides a set of candidate sites for the file, which are located in different domains (geographical area). It is assumed that two locations within the same domain have the same storage and transfer costs. The set of sites at which the file is replicated is selected from the candidate sites using a heuristic algorithm that maximizes the difference between the replication benefits and replication costs.

The availability checks are performed at a variable time interval, based on the results of previous checks. For instance, if during the last checks more replicas are needed, the frequency is increased.

One weakness of the algorithm is that each site makes its own replication decision when file availability decreases. Therefore, extra replicas may be created by several sites, when

only one is needed. This is a price that is paid in order to avoid centralized control. Another drawback is that, as long as the availability conditions are met, the data reallocation is not performed. This is an inefficient approach because the access to the files might change over time. Also, this strategy deals with files which do not have correlations between them as in the case of database fragments.

A Caching Algorithm in Tree-Shaped Networks

Since the caching algorithms deal with the Web objects and Web objects are considered individually, none of the caching algorithms can be used in our case.

In [14] the authors propose a strategy in which the response time is the determining factor for allocation. The goal is to minimize the number of replicas while meeting the clients' QoS (quality of service) and server capacity constraints. Each client has to maintain an access latency that is less than a specified threshold. This approach is different from others like [49, 50, 17], which minimize the communication cost but may fail to meet the requirements of all clients. In the previous approaches, if a user rarely accesses a replica, no copy is placed closer because the read gain does not cover the update cost. Users that are clustered have a local copy and are served faster than they need to be. However, those residing outside the clusters experience significant delays. The caching algorithm proposes a method to satisfy every user's QoS requirements. Replicas of objects are placed such that each user can get a response in a bounded time interval. If a new client accesses an object, a copy of it is placed in the client's proximity. The clients are the leafs of the tree and the data source is the root. The algorithm pushes the replica as high as possible in the tree while still maintaining the QoS of the leaf nodes. This way, the maximum number of clients can be served using the minimum number of replicas. Also, the server resources are taken into consideration and objects are not placed on sites with low resources. The proposed caching algorithm is good for Web content dissemination, as most Web objects are rarely written. However, in distributed database environments, enforcing a bounded response time for each user would be too expensive if data is updated frequently.

A Caching Algorithm in General Topology Networks

A bandwidth-constrained dynamic allocation strategy is examined in [47], with a focus on WAN environments. The goal is to place objects at distributed hierarchical caches to minimize the clients' access time to those objects, subject to bandwidth constraints at each cache. The system is modelled as a set of N distributed cache machines and a set of S origin servers connected in a network. The clients access a set of M shared objects maintained at any of the S servers and cached at the cache machines. The objects have

an expiry time after which they cannot be considered useful and are removed from cache. Each cache machine has a fixed available bandwidth bw that is used to pull objects into its cache. A client requests an object, say α, from a caching machine i and, if α is local, the client is served immediately at no cost. Otherwise, cache i gets the object from the closest cache j at cost $c(i, \alpha)$, serves the client, and stores the object locally. $c(i, \alpha)$ is defined based on the communication cost between caches i and j. In the unfortunate case when none of the N cooperating caches has a copy, the cache machine takes the object from the origin server with a higher cost Δ. The placement has to be completed within a time period, t_{fill}. It is shown that, if the value of t_{fill} is too small, objects might be allocated in suboptimal locations. On the other hand, if the value is too large, the system has a high transient miss rate. The miss rate occurs because client reads are performed in parallel with data placement. Given the probability that a site accesses an object and the available bandwidth to push the objects into caching machines, the algorithm finds an allocation that minimizes the access cost and is accomplished in t_{fill}. Since the bandwidth is the constraint, there is no guarantee that all clients receive a response to their requests in a bounded time interval. The algorithm does its best to minimize the access time, but it can never be as efficient in serving the users as the algorithm proposed in [14] in which more bandwidth is consumed. It is always a trade-off: buy more bandwidth or expect a longer wait time.

The presented strategies that deal with database environments fail to consider the relations between data fragments, although fragments may be accessed together. On the other hand, strategies that might work well in Web dissemination cannot be used in the database environment. To conclude, the dynamic distribution problem needs further research.

Chapter 3

The Spring Algorithm

In this chapter we propose an incremental allocation algorithm, called the *Spring Algorithm* that minimizes the overall access cost to fragments. The name derives from the observation that the DDBMS can be modelled as a system of springs where nodes corresponding to queries and fragment replicas are connected to each other by springs (edges), and where a spring represents the affinity of the two nodes it connects. We first describe the distributed system environment, then introduce the spring system as an analogy to it and describe the algorithm.

3.1 Environment Description

The distributed system we analyze has three components: (a) the *sites* which store data or from which queries are executed, (b) the *data* (at the granularity of relation fragments) accessed by queries, and (c) the *network links* which connect sites. Based on some factors that are discussed in detail in Section 3.1.3, each network link has a cost associated to it. We present these components next and, in the last part of the current section, we focus on queries which make the environment dynamic.

3.1.1 Sites

The sites cooperate to improve the DDBMS's performance by executing system tasks such as data replication, data movement, and data deletion. The users connect to one of the sites close to their location to execute a query. It is not necessary that this site hosts any of the data the user has required, as it cooperates with the other sites to serve the user. Each site has limited storage capacity. Throughout the chapter, sites are denoted with S.

3.1.2 Data fragments

As indicated, we work with fragment-level granularity, and use as size metric the kilobyte (KB). Throughout the chapter we use F as notation for a fragment and $R_k(F_i)$ for a replica k of fragment i. For simplicity, we use *replica* to indicate a replica of a fragment. The site $S = site(R_k(F_i))$ represents the location of $R_k(F_i)$, and $size(F)$ represents the size of the fragment, in KB.

The fragments with high reads are good candidates for replication as the benefit from reading data locally is high. On the other hand, the fragments with high updates should be located at a single site in order to avoid paying consistency costs. In a highly dynamic environment such as the Internet, not only does the access pattern to fragments change over time, but the nature of access to fragments changes as well. A fragment that was mostly read (*read intensive*) could change and become *update intensive*. Also, a part of a fragment could change its access nature and, to deal with these situations, re-fragmentation should be considered. However, this is a problem orthogonal to ours, and we do not study it any further. We assume the fragmentation has already been performed.

3.1.3 Network Links

Each link between a pair of sites has a communication cost associated to it. Since in our case communication takes place in an Internet environment, the access cost is variable. Based on the network load and the number of nodes that are crossed to reach the destination (the path is not always the same), the response time can vary. However, the calculation of the access cost is outside the scope of this thesis, and we assume the cost is known. The access cost $AC(S_i, S_j)$ between sites S_i and S_j represents the "distance" between them (the average delay of sending 1KB of data between two sites, measured in ms/KB). The access cost could be made more comprehensive by using a more complicated function to compute it. We consider $AC(S_i, S_i)$ to be zero, and make the simplifying assumption that the cost between two sites is symmetric.

3.1.4 Data Access Framework

We have presented the environment which defines the distributed system: the fragments are located at sites, and sites have distances between them. In this section we introduce the queries (denoted with Q) which access the fragments. They represent the factors that cause fragment reallocation, as the goal is to minimize the cost of executing them. For instance, a replica that is never accessed by queries does not move, as no constraint is forcing it to move. Also, two replicas of a fragment are not pushed towards each other (or

even merged) to avoid consistency costs unless they are updated.

A distributed database system is constantly accessed by queries. The access pattern of a fragment is determined by analyzing the characteristics of the queries accessing the fragment over a time interval. Similarly, for the whole system, we analyze only the set of queries (denoted $\mathcal{Q}$) that ran on the system in a time interval $[t_{start}, t_{stop}]$ and consider them relevant for the general access to fragments[1]. Optimizing fragment allocation for those queries would actually optimize access in general. When choosing the time interval, there is a tradeoff between the length of the time interval and the accuracy of the captured access pattern. If the interval is long, it will capture the access characteristics better than a short one as the set of executed queries is larger. However, a smaller time interval means that the allocation is performed more often, therefore the system performance is better. Choosing the proper interval depends on the application that uses the DDBMS. If the system is heavily accessed, the time interval should be shorter than the interval chosen for a system accessed infrequently. We consider the set of queries $\mathcal{Q}$ and the sites on which they were executed to be known. We expect the changes in the access pattern to be incremental, therefore $\mathcal{Q}$ changes gradually. The allocation is performed if the access pattern to fragments has changed in the last interval, otherwise the fragment configuration remains unchanged.

Some of the queries may be executed more than once in the time unit, therefore they can be found multiple times in the set $\mathcal{Q}$ of executed queries. We define the frequency of the query $freq(Q)$ as the number of times query Q is found in $\mathcal{Q}$.

The goal of the allocation algorithm we propose is to minimize the overall access cost of queries to fragments. This is accomplished by relocating fragments in order to better serve the queries.

We define $\mathcal{Q}_{R_k(F_i)} \subseteq \mathcal{Q}$, as the set of queries that access the replica $R_k(F_i)$. Site $S = site(Q)$ refers to the location at which query Q runs. For a select statement, the selectivity of a query $sel(Q)$ is defined as the ratio between the number of tuples that satisfy the query and the number of tuples in the fragment. These notations are used throughout the chapter.

3.2 Cost Constraints

To get good system performance, data should be close to the points of access to reduce the communication cost. Our interest is in the aggregate access cost, which is computed as the sum of the costs of running all the queries in the system. We adopt the query execution cost formula given in [41]:

[1]These are traces of query executions, a snapshot of the system over a time interval.

$$Time = \underbrace{T_{CPU} \cdot \#insts}_{CPU\ cost} + \underbrace{T_{I/O} \cdot \#I/Os}_{I/O\ cost} + \underbrace{T_{MSG} \cdot \#msgs + T_{TR} \cdot \#bytes}_{Comm.\ cost}$$

The first component measures the local processing time: T_{CPU} is the time of a CPU instruction and $T_{I/O}$ is the time of a disk I/O. T_{MSG} is the fixed time of initiating and receiving a message, while T_{TR} is the time it takes to transmit a unit of data from one site to another.

In this thesis, our goal is the minimization of the overall communication cost, because, in our case, we consider it to be the largest of the three. Because of this goal, each individual communication cost becomes a constraint. Based on the communication characteristics, there are three types of constraints (detailed in the next sections):

- the *query access constraints* force the replicas to get closer to the users accessing them;
- the *fragment correlation constraints* force replicas that are accessed together to stay close to each other;
- the *replication constraints* force replicas of fragments to stay close to each other (or be removed entirely) to avoid high update costs.

These constraints determine whether replicas change their present location due to query access. As replicas of different fragments or replicas of the same fragment interact, if some of them move, replicate, or merge (two copies of the same fragment at the same site), others may do the same. In order to achieve the goal of reducing the overall cost, all of these constraints must be considered together (more details are presented in Section 3.4.3).

3.2.1 Query Access Constraints

The *query access constraints* are illustrated in Figure 3.1(a). Each constraint is defined as the cost incurred by a query to access the replica it references. In order to minimize this cost, each query pulls the replica towards its location.

The query access cost may significantly vary depending on the size of the transferred data, and on the site from which the query accesses that data. Formally, we define the **query access constraint** given by the access of Q to $R_k(F_i)$ as

$$\mathcal{T}^{Q}_{quer}(R_k(F_i), Q) = AC(site(R_k(F_i)), site(Q)) \cdot size(D_{R_k(F_i),Q}(Q)) \cdot freq(Q)$$

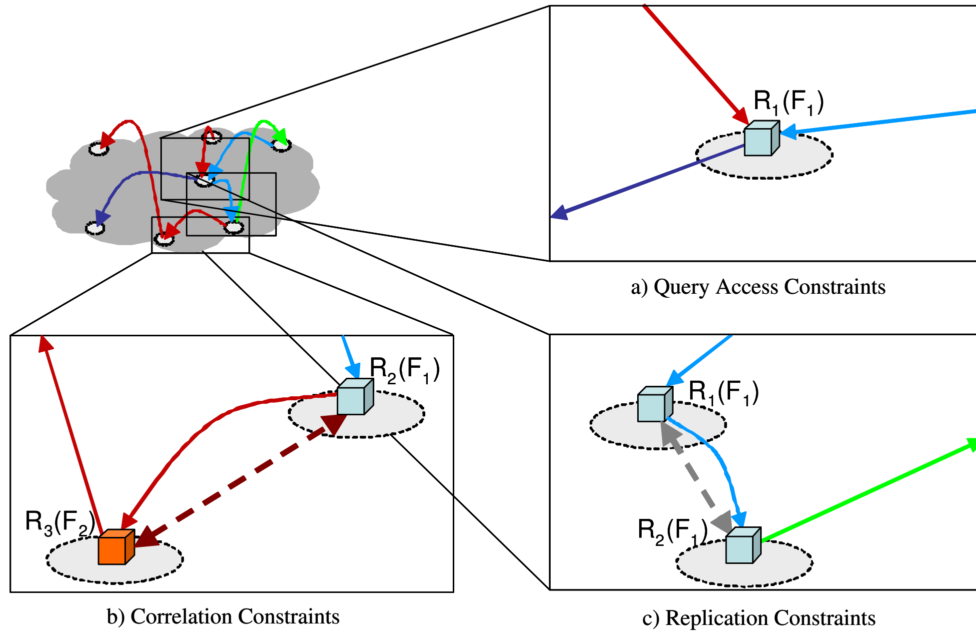

Figure 3.1: Cost constraints

where $D_{R_k(F_i),Q}(Q)$ is the data transferred due to query Q between the site where $R_k(F_i)$ is stored and the site where Q runs. In the formula, $site(Q)$ influences the cost by its proximity to the data source. A longer distance intuitively means a higher cost, although link characteristics may also influence it. The cost also depends on the size of $D_{R_k(F_i),Q}(Q)$: the more data, the higher the cost.

This constraint "pushes" $R_k(F_i)$ closer to the location of Q. However, $R_k(F_i)$ should be placed on a site that will minimize the sum of the costs to access it, not only the cost that Q pays.

Figure 3.1(a) shows the case of an *independent fragment*, which is a non-replicated replica that has only query access constraints. If a select statement is executed, assuming all tuples have the same size, the size of the data transferred to the site running the query is $size(D_{R_k(F_i),Q}(Q)) = size(F_i) \cdot sel(Q)$, i.e. it depends on the size of the accessed fragment F_i, and the selectivity of Q. The directions of the arrows show the way the data was transferred when the query was executed (remember we have a snapshot of the system over a time interval). In the figure, there are two updates and one read.

3.2.2 Correlation Constraints

Correlated replicas are those that belong to different fragments and that have execution interdependencies between them (see Figure 3.1(b)) such that the movement of one replica may affect others. This could be due to, for instance, *join operations*. To minimize the network cost, the small replica could be sent to the bigger replica[2] when executing the query. The direction of data transfer when the join was executed is reflected in the direction of the arrow.

We define the **correlation constraint** between two replicas that was produced by a query Q as

$$\mathcal{T}_{corr}^{Q}(R_k(F_i), R_l(F_j)) = AC(site(R_k(F_i)), site(R_l(F_j))) \cdot size(D_{R_k(F_i),R_l(F_j)}(Q)) \cdot freq(Q)$$

where $size(D_{R_k(F_i),R_l(F_j)}(Q))$ is similar to the previous case – the size of the data transferred due to query Q between the two sites storing the two replicas. Assuming that the small replica is the one that is shipped,

$$size(D_{R_k(F_i),R_l(F_j)}(Q)) = min(size(R_k(F_i)), size(R_l(F_j))).$$

This constraint "pushes" two correlated replicas to the same site, to minimize the overall cost. If replicas are located at the same site, the correlation cost is zero.

When allocating data, the correlation constraints must be considered together with the query access constraints. The problem of cost minimization is more complicated than in the case of independent replicas. Moving a replica $R_k(F_1)$ alone to minimize the access cost to it could increase the access cost of another replica $R_l(F_2)$ which is correlated with $R_k(F_1)$. If the *gain* from moving $R_k(F_1)$ is smaller than the *depreciation* from accessing $R_l(F_2)$, the system should not change its configuration, or should consider moving $R_l(F_2)$ as well.

3.2.3 Replication Constraints

Whenever a query is served from a nearby copy of a fragment, the cost to access it is small, as illustrated in Figure 3.1(c). However, depending on the update frequency, having a copy locally may not be wise if the cost of keeping it consistent is higher than the benefit of reading it locally.

The cost of maintaining consistency of a replica depends on: (a) the replica consistency mechanism and the way the updates are propagated (captured by the direction of the

[2] To improve the transmission cost, sometimes semi-join operators are used. This case overly complicates the problem and therefore it is not analyzed. Our goal is to present a method for fragment replication and reallocation, therefore query optimization is not a concern.

arrows), (b) the fragment's update frequency, and (c) the sites at which the replicas are located. A high number of updates or sparse placement of replicas incurs a high update cost. This constraint "pushes" two replicas to the same site.

The **replication constraint** between two replicas of a fragment is defined as

$$\mathcal{T}^{Q}_{repl}(R_k(F_i), R_l(F_i)) = AC(site(R_k(F_i)), site(R_l(F_i))) \cdot size(D_{R_k(F_i),R_l(F_i)}(Q)) \cdot freq(Q)$$

where $D_{R_k(F_i),R_l(F_i)}(Q)$ represents the size of the update message sent due to query Q between the two replicas.

The consistency mechanism is not an issue in this thesis, as we look at execution traces.

All cost constraints are expressed in milliseconds as the access cost was defined in ms/KB, the data size in KB, and the frequency does not have a unit. The goal is to minimize the sum of all these constraints, and there is no bound on the time it takes to run any $Q \in \mathcal{Q}$ in order to obtain the overall minimum. A bound would make the problem more difficult. For instance, replicas may still have to be created even though the fragment is only updated.

3.3 The Spring System Approach - An Analogy

We model data replication and redistribution as a "Spring System" and we show next that the spring system simulates the problem we want to solve. The three constraints presented in Section 3.2 are presented as spring forces between objects. The spring system components are presented in the following three sections.

3.3.1 The Objects

Objects are used to represent replicas and queries in the spring system. Replicas that were defined in Section 3.1.2 are mobile objects in a spring system, whereas the queries are fixed. The **replica objects**, presented in Section 3.1.2 as replicas, are pulled in different directions by springs. The resulting combination of the forces determines the position of the replica objects, which will create a balanced system. The **query objects** represent fixed objects that pull the replica objects they access towards them. A query object is located on the site where the query runs.

3.3.2 The Springs

The spring characteristic that interests us most is stretchability. Whenever the spring is taken out of its usual state, there is a force that wants to pull it back. The *springs* are used

to represent the *constraints* presented in Section 3.2. For reasons which will be presented later, we assign *directions* to springs. They have the same significance as the arrows, i.e. the direction the data is transferred when the query was executed. In Figure 3.2 we show the springs that correspond to the constraints presented in Figure 3.1:

- The **query spring** – saw tooth edge in Figure 3.2(a) – connects a query object to a replica object. It creates an energy[3] that pulls the replica object towards another location that favors the query which created the spring (because the replica object is mobile and the query object is fixed). This energy was defined in Section 3.2.1 as the cost $\mathcal{T}^{Q}_{quer}(R_k(F_i), Q)$. When the replica object is located at the site on which the query is running, the energy is zero (because the access cost component AC is zero).
- The **correlation spring** – square tooth edge in Figure 3.2(b) – connects two replica objects of different fragments (for instance due to join operations). This spring creates an energy that brings the objects closer to each other (to reduce the cost to execute the join). This energy was defined in Section 3.2.2 as the cost $\mathcal{T}^{Q}_{corr}(R_k(F_i), R_l(F_j))$. The closer the objects are, the lower the energy in the spring, due to the access cost component. When the replica objects are at the same site, the energy is zero.
- The **replication spring** – sinusoidal edge in Figure 3.2(c) – connects two replica objects of a fragment F_i. Since having replicated fragments results in a cost to maintain the consistency, the goal of this spring is to push two replica objects of a fragment close to each other, in order to reduce the energies in the springs. The energy was defined in Section 3.2.3 as the cost $\mathcal{T}^{Q}_{repl}(R_k(F_i), R_l(F_j))$. We will later show that when two replicas are on the same site, one is removed.

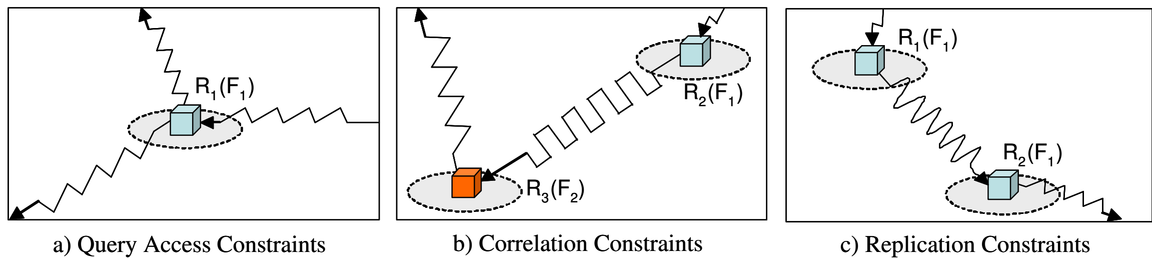

Figure 3.2: Analogy between the distributed database system and the spring system

[3]In physics, forces always create energies.

3.3.3 The Sites and The Network Links

A snapshot of the dynamic processing that takes place in a DDBMS environment over a time interval can be modelled as in Figure 3.3. The lower plane represents the physical system depicting the sites and the network links which were presented in Sections 3.1.1 and 3.1.3. The upper plane depicts the spring system[4], with objects and springs.

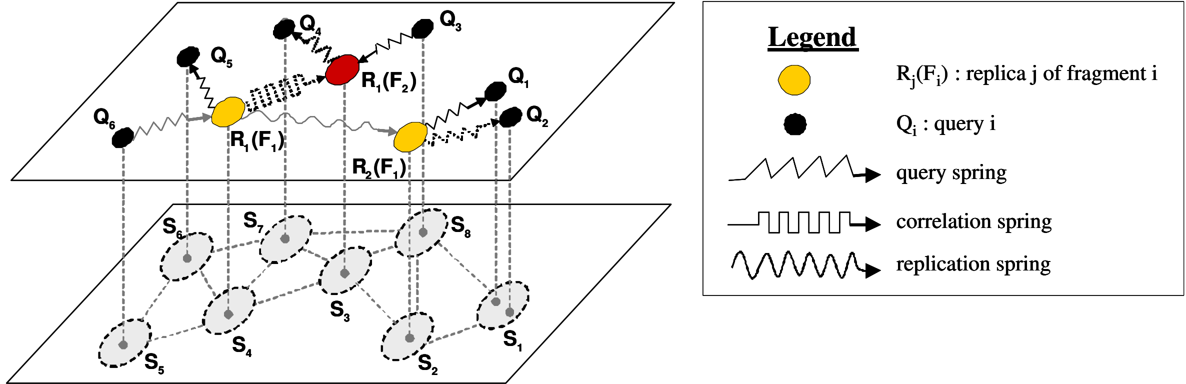

Figure 3.3: The two layers of the problem

The projection to the physical layer represents the implementation of function $site(R_k(F_i))$ or $site(Q)$, which were previously introduced. In the figure, for example, each replica object and query object are located at different sites, except for queries Q_1 and Q_2 which are both located at site S_1.

The figure presents an equilibrium state in which the objects are not moving. However, if the access pattern has changed and, in the new set $\mathcal{Q}$, queries Q_1 and Q_2 no longer exist, the replication spring may move $R_2(F_1)$ closer to $R_1(F_1)$ by placing it at site S_3 and then S_4 to reduce the overall cost[5] to access the replicas. Also, if the join query Q_4 no longer runs, $R_1(F_2)$ may be moved to S_8 for the same reason.

3.3.4 System Equilibrium

As presented, springs create energies that may move a replica object. In Figure 3.4, a change of the force $\vec{f_2}$ acting on an object hooked with springs moves the object to a new location to balance the effect, until $\sum \vec{f_i} = 0$. We call this the *equilibrium state*. The total energy of the object is defined as $W_t = W_c + W_p$, where at equilibrium the kinetic

[4]Springs with the same line pattern represent one query.
[5]The overall cost is defined in following section as the energy of the spring system.

energy $W_c = 0$ as the object does not move, and the potential energy W_p is minimal. At equilibrium, $\sum \vec{f_i} = 0 \Leftrightarrow min(W_p)$. Because in the spring system the connected objects interact, to obtain an overall stable system all of them must be in a state of equilibrium.

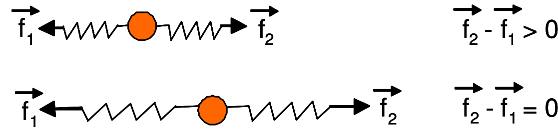

Figure 3.4: Object connected with springs

Because, in physics, the **spring system energy** at equilibrium is defined as the sum of all energies in the springs and, in DDBMS, spring energies represent costs, minimizing the sum of all energies in the springs corresponds to the minimum overall access cost in the DDBMS. *The locations of the replica objects at equilibrium will give the locations where the replicas should be placed in the DDBMS.*

To give a more precise picture of the equilibrium state, in the system illustrated in Figure 3.5, only if the change of the resulting force $\vec{f_r}$ acting on the object $R_1(F_1)$ is significant enough to beat the friction force f_f, is $R_1(F_1)$ redirected to a new location. Otherwise $R_1(F_1)$ must be in the optimal neighborhood, denoted with a circle in Figure 3.5(b). In the DDBMS, the replicas are pushed towards the optimal locations but they will stop as soon as they get into the optimal neighborhood.

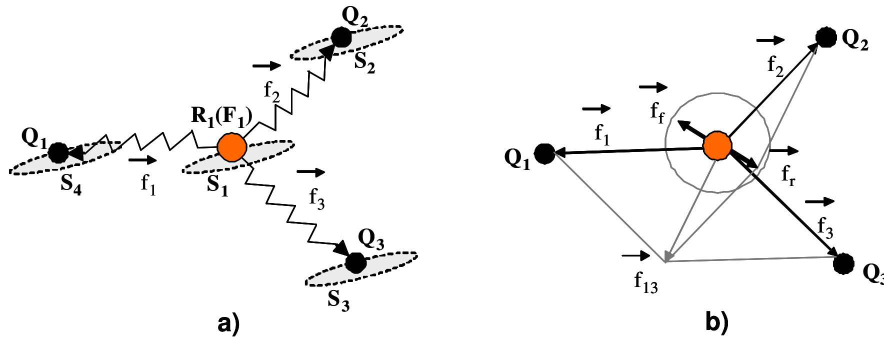

Figure 3.5: $\vec{f_r}$ should be greater than the $\vec{f_f}$ to move the object.

In the figure, the performance of the replica object placement is bounded by the energy created by the friction force[6]. This energy is defined in Section 3.4.5 as the cost to move

[6]We later show that in the DDBMS this is not the only factor influencing the performance.

the replica. If there is a benefit from moving the replica but it does not cover the cost to move it, the replica stays at the current location.

3.4 The Spring Algorithm

Before presenting the Spring Algorithm in detail, four concepts have to be introduced: the spring system represented as a graph, the bubble, the cost problem, and the replicate, move, and delete techniques. These are all used when presenting the algorithm in Section 3.4.5.

3.4.1 Graph and Placement

The spring system illustrated in the top plane in Figure 3.3 can be represented as a weighted directed graph[7] $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ with two types of vertices and three types of edges. The set of vertices $\mathcal{V}$ represents the union of all replica objects and all query objects. The edges $\mathcal{E}$ represent all springs in the spring system. The edge directions are the direction of data transfer, and the edge weights $\mathcal{W}$ as the energies in the springs, as discussed earlier.

The lower plane which contains information about network links, sites, fragments, and query allocation is kept in a data structure referred to as placement ($\mathcal{P}$). The full information about the environment is therefore captured in the pair $(\mathcal{G}, \mathcal{P})$.

The pair $(\mathcal{G}, \mathcal{P})$ is fed into the Spring Algorithm and the output is a new graph-placement pair $(\mathcal{G}', \mathcal{P}')$. The graph $\mathcal{G}'$ could have a different number of vertices since in the new configuration some fragments could have fewer or more replicas. Also, the number of edges could be different because, if there are more replica objects, there are more replication springs that propagate the updates. If replica objects move or replicate, the change in their location is reflected in the placement $\mathcal{P}'$. The way the algorithm performs the changes on the initial pair $(\mathcal{G}, \mathcal{P})$ is presented in Section 3.4.5, where we describe the Spring Algorithm.

We define the **energy of a replica object** ($W_{R_k(F_i)}$) as the sum of all energies in the outward springs connected to it. We gave directions to the springs in order to avoid double counting edges between objects. As an exceptional case, in order to give energies to replica objects only, we count the energy of the query springs that update replica objects towards the replica objects. The energy of the spring system, which was defined as the sum of all energies in the springs, can now be formulated as the energy of all replica objects in the system. Each replica object stores the value of its own energy and, unless the springs that connect to the objects change, their energies are not recomputed.

[7] We model it as a graph in order to simplify the incremental computation of the energies.

To show an example of why this is useful, let's consider the case illustrated in Figure 3.6 (for simplicity no sites are shown). Moving a replica object may cause other replica objects connected to it to do the same. However, the farther they are from the moved replica object, the less the shockwave (the change in energy) is felt. Only the energies of the replica objects which have springs in common with the replica objects that were reallocated are recomputed.

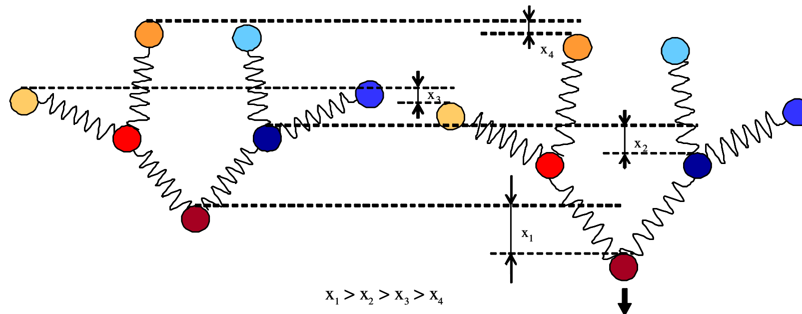

Figure 3.6: A reallocation affects more the closer replica objects.

3.4.2 Bubble Definition

As mentioned before, objects can be correlated. If moving a replica object $R_k(F_i)$ decreases its energy but increases the energy of a correlated object $R_l(F_j)$ by a greater amount, then $R_k(F_i)$ should not be moved. For instance, moving a replica object $R_k(F_i)$ closer to a set of query objects that accesses it reduces the cost queries pay, but increases the cost the other replica objects pay to propagate the updates and keep $R_k(F_i)$ consistent. The replica objects with affinities to each other should therefore be analyzed together, not individually.

We **define the bubble** a set of nodes from $\mathcal{G}$ such that (1) *there is an undirected path between each pair of nodes in the bubble*, and (2) *there are no paths between the nodes in the bubble and the nodes outside the bubble*[8].

The bubble is the **unit of information** that needs to be considered when changing any object's allocation. As a particular case, the independent replica object is the only object in the bubble: $Bubble(R_1(F_i)) = \{R_1(F_i)\}$.

Because the bubbles are independent of each other, minimizing the energy of each bubble results in minimizing the overall energy of the spring system. The bubbles can be analyzed in parallel, therefore *from now on we only refer to one bubble.*

[8]In this thesis we do not consider the interesting case where the bubbles are interconnected.

3.4.3 The Bubble Energy

For easy reference, from now on we use the pair $(\mathcal{G}, \mathcal{P})$ to define only the bubble we analyze, not the whole environment. The goal is to find the minimum overall energy of the spring system defined by $\mathcal{G}$. Also, we have shown that the unit of information that needs to be analyzed when changing the system configuration is the bubble.

We define the **total energy of a bubble** (W_{bubble}) in the spring system as the sum of all springs energies. We introduce them in four cases, in the increasing order of their complexity. The easiest case is one in which the bubble is created by an independent replica object. Here, $\mathcal{G}$ contains query and replica objects and query springs. In the second case there are correlated replica objects, in which no fragments are replicated. $\mathcal{G}$ contains the same components as the first case, plus correlation springs. Cases three and four allow replication. In the third case, the bubble is composed only of replica objects of the same fragment. $\mathcal{G}$ contains the same components as the first case, plus the correlation springs. The fourth case is the most complex and the most probable to occur. It contains all five components, the two types of nodes and the three types of springs. The following sections analyze each of these cases, and show how the energies are computed. At the end we give a general algorithm that incrementally computes the energy of the bubble, which is used by the Spring Algorithm.

Case 1 - Uncorrelated Fragments, with no Replication

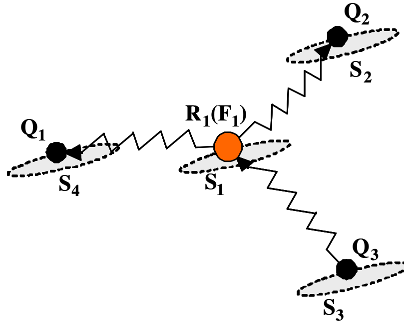

Figure 3.7: All fragments are non-replicated and uncorrelated

This case is depicted in Figure 3.7. The replica object $R_1(F_1)$ located on site S_1 is independent, therefore minimizing the energy of the replica object minimizes the energy of the bubble. Since there are only query springs hooked to the objects, the sum of all

energies created by these springs comprises the total energy. For this particular case, $W_{bubble} = W_{R_1(F_1)}$. In a general case, the bubble energy given by the query springs is defined as

$$W^{quer}_{bubble} = \sum_{\forall R_k(F_i) \in bubble} \; \sum_{\forall Q \in \mathcal{Q}_{R_k(F_i)}} \mathcal{T}^{Q}_{quer}(R_k(F_i), Q)$$

where, in Case 1, $W_{bubble} = W^{quer}_{bubble}$. The notation W^{quer}_{bubble} is used for further reference. Recall that, we only gave energies to replica objects, not to query objects.

Case 2 - Correlated Fragments, with no Replication

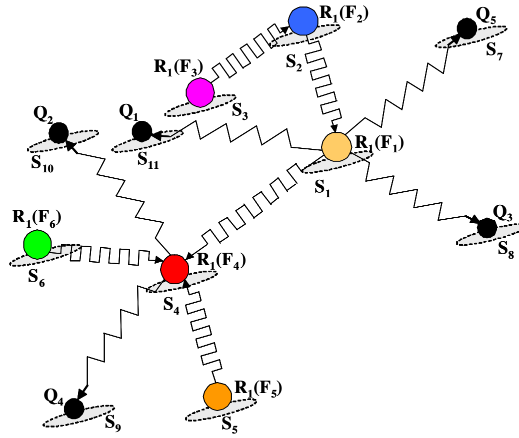

Figure 3.8: Non-replicated but correlated fragments.

This case is illustrated in Figure 3.8. The energy of the bubble is given as the sum of query springs energies and correlation springs energies. To reduce the bubble energy, both types of springs must be considered together. In the general case, the bubble energy from correlation springs is

$$W^{corr}_{bubble} = \sum_{\forall R_k(F_i) \in bubble} \; \sum_{\forall R_l(F_j) \in bubble} \; \sum_{\forall Q \in \mathcal{Q}_{R_k(F_i)} \cap \mathcal{Q}_{R_l(F_j)}} \mathcal{T}^{Q}_{corr}(R_k(F_i), R_l(F_j)), i \neq j.$$

In Case 2, the bubble energy is computed as

$$W_{bubble} = W^{quer}_{bubble} + W^{corr}_{bubble}.$$

Reallocating a replica object will reset the energy of others replica objects in the system as well. For instance, in the figure, the reallocation of $R_1(F_4)$ affects the energy of $R_1(F_1)$.

Case 3 - Uncorrelated Fragments, with Replication

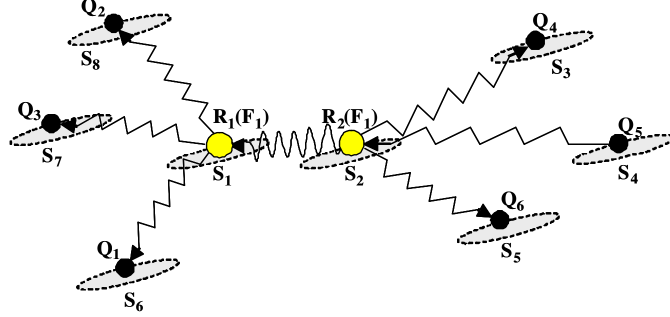

Figure 3.9: Replicated but non-correlated fragments

This case is illustrated in Figure 3.9 and is similar to Case 2 – instead of correlation springs there are replication springs.

In the general case, the bubble energy from replication springs is given by

$$W_{bubble}^{repl} = \sum_{\forall R_k(F_i) \in bubble} \sum_{\forall R_l(F_j) \in bubble} \sum_{\forall Q \in \mathcal{Q}_{R_k(F_i)} \cap \mathcal{Q}_{R_l(F_j)}} \mathcal{T}_{repl}^{Q}(R_k(F_i), R_l(F_j)), i = j.$$

In Case 3, the bubble energy is computed as

$$W_{bubble} = W_{bubble}^{quer} + W_{bubble}^{repl}.$$

Case 4 - Correlated Fragments, with Replication

This case is illustrated in Figure 3.10(a). The energy is computed as the sum of the energies in all springs in the system:

$$W_{bubble} = W_{bubble}^{quer} + W_{bubble}^{corr} + W_{bubble}^{repl}.$$

The bubble energy is computed incrementally, as presented in Algorithm 2 (called *GetBubbleEnergy*) on page 42 of this chapter. The energy of a replica object can be at

Figure 3.10: Replicated and correlated fragments

least zero (zero if all accesses to it are from the same site that stores it, as the access cost component is zero). However, the Spring Algorithm that uses *GetBubbleEnergy*, can reset the energy of an object that changed its allocation configuration, by setting the energy of the object to $W_{R_k(F_i)} = -W_{R_k(F_i)}$. This way we know that the energy of the object has to be recomputed (because it is negative), but still keep the old value of the energy.

In lines 4 to 10, we create the set $\mathcal{C}$ used to store the replica objects that were affected by changes performed by the Spring Algorithm. Also, from the old bubble energy, we remove the energies of the replica objects that no longer apply. Then, the algorithm recomputes the bubble energy and the object energies. This is accomplished by adding the energies in the query springs (lines 12-17), correlation springs (lines 19-24), and replication springs (lines 26-31) that have changed. At the end, the current energy of the bubble is saved, and the result is returned.

3.4.4 Replication, Move, and Delete Techniques

Changing access patterns may necessitate the reallocation of replica objects in order to improve query performance. This section defines the techniques used in the Spring Algorithm to simulate movement or replication of replicas in DDBMS.

The next two methods are used by the Spring Algorithm (presented in Section 3.4.5):

- *SimulateMove*($R_k(F_i), S'$) is used to find whether a replica object $R_k(F_i)$ should be

moved to S'. This method virtually reallocates $R_k(F_i)$ to site S' (remember the projections in Figure 3.3). The Spring Algorithm then recomputes the energy of the bubble. If the energy in the virtual configuration is lower than the initial energy, it makes sense to move the $R_k(F_i)$ to S'.

- *SimulateReplicate*$(R_k(F_i), S')$ finds if there is a site S' on which $R_k(F_i)$ should be replicated for better performance. It is similar to *SimulateMove*, but instead of virtually reallocating $R_k(F_i)$ to site S' it creates a copy of $R_k(F_i)$ on site S'.

When a fragment is replicated in the DDBMS, the corresponding replica object divides like an organic cell creating two objects identical to the original one. Figure 3.11 shows how an object is replicated to a neighbor site because the forces pulling from the replica object created an energy that "broke" the object.

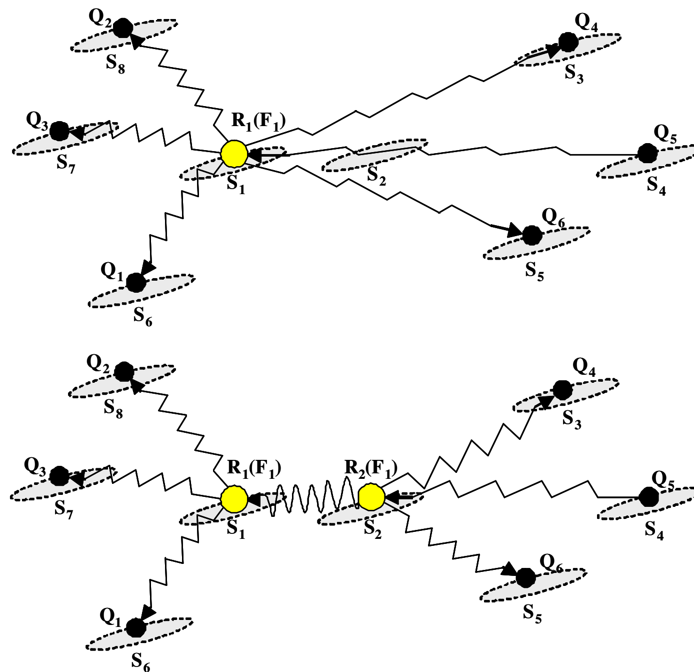

Figure 3.11: Replication step in the Spring System

In order to simulate the situation in Figure 3.11, after the new replica is virtually created, the springs have to be reconnected. Otherwise, the springs remain connected to $R_1(F_1)$. Reconnecting springs after each replication simulation (and there are many) increases the running time of the algorithm.

When a replica is moved in the DDBMS, the corresponding replica object moves together with the springs connected to it. For example, Figures 3.10(a) and (b) show how an end of the spring connected to $R_1(F_2)$ is still connected to the replica object, even though replica object's location has changed. However, after a replica is reallocated, all springs connected to the fragment (not only that replica) should be reanalyzed to see if there is a closer replica to connect to. In Figure 3.12, after $R_2(F_1)$ is moved from S_2 to S_3, the query object Q_2 should connect to $R_1(F_1)$ which is now closer.

For the move simulation, we chose not to connect the springs as the process of reconnecting springs is extremely time demanding due to of the large number of springs. Also, it is not critical[9] to reconnect the replicas for each simulation as in the case of replication simulation. To summarize, spring reconnections are performed when a replica object is moved or replicated, and when the replica object is virtually replicated.

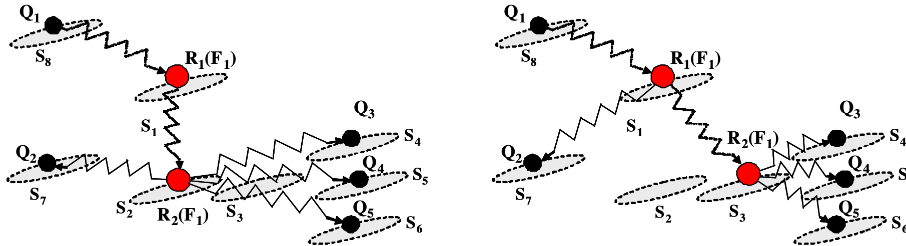

Figure 3.12: After $R_2(F)$ is moved, Q_2 uses $R_1(F)$.

After the move is performed, if a replica is to be placed on a site that already has one, that replica is **deleted**. All the springs that were connected to the removed replica are now connected to the remaining one. Note that, the deletion of a replica is performed only when a replica is actually moved to a site that already has one, not when it is virtually allocated to that site for testing reasons.

3.4.5 The Spring Algorithm - Description

The Spring Algorithm (described in Algorithm 5) is incremental algorithm that performs the replication and redistribution of fragments based on an initial configuration. We call it incremental as the energy of the bubble is computed incrementally.

Before presenting the Spring Algorithm let's describe the six helper methods it uses:

[9]In Chapter 4 we show that this has an impact on performance in small settings.

- *GetBubbleEnergy(bubble)*, as a reminder, is a method that returns the energy of the bubble as the sum of the energies of all replica objects in the *bubble* (all springs in the system), and was presented in Algorithm 2.

- *GetClosestSites(x, S, size(F_i))* is a method that returns the x closest sites (in terms of access cost) to site S. These sites are willing to store data from S and have enough space available to accommodate a replica object of size $size(F_i)$. In order to avoid repetitions, we assign the size of fragments to F_i, not to each $R_k(F_i)$.

- *SimulateMove($R_k(F_i)$, S)* was detailed in Section 3.4.4. It virtually moves $R_k(F_i)$ to S so that the bubble energy for the new allocation can be recomputed.

- *SimulateReplicate($R_k(F_i)$, S)* was detailed in Section 3.4.4. It virtually replicates $R_k(F_i)$ at S so that the bubble energy for the new allocation can recomputed.

- *CheckMove($R_k(F_i)$, W_{init})*, defined in Algorithm 3, finds a better location for a replica object, based on the fixed location of the other objects in the bubble. It takes as parameters the object $R_k(F_i)$ that is being analyzed, and, for comparison reasons, the initial energy of the bubble W_{init}. This function finds the best site among the neighbor sites (line 3) where the object should be located. The best site found (line 8) and the bubble energy (line 9) in the simulated configuration are returned to the $SpringAlgorithm$.

- *CheckReplicate($R_k(F_i)$, W_{init})*, presented in Algorithm 4, works as $CheckMove$ with the difference that it simulates that the given parameter $R_k(F_i)$ is replicated at S and is also kept at the present location. Note that neither *CheckMove* nor *CheckReplicate* change the physical allocation.

The Spring Algorithm gets as input the pair ($\mathcal{G}$, $\mathcal{P}$) describing the system and a queue of replica objects to be analyzed ($\mathcal{L}$). A replica object is placed in $\mathcal{L}$ if its access pattern changed. The Spring Algorithm returns another pair ($\mathcal{G}'$, $\mathcal{P}'$) such that, in the new configuration, the bubble energy is reduced.

A replica object $R_k(F_i)$ is extracted from the queue to be analyzed (line 1). If the fragment allows movement (this property is associated with all replicas of a fragment), the Spring Algorithm tries to find a better location such that the difference between the energies in the original and the final configuration is greater than the move threshold σ_M. If a location is found, $R_k(F_i)$ is moved[10] and then is reintroduced in $\mathcal{L}$ to be reanalyzed

[10] Remember, the replica is deleted if there is already another replica at that site.

(lines 3-13). Next time it is analyzed the process is repeated, but now the replica is at a different site.

If the replica object is at the best location found by the algorithm and the fragment allows replication (this property is associated with all replicas of a fragment), the Spring Algorithm tries to get an even lower energy from replication. If a new replica object is created, the algorithm reintroduces both objects in the queue to be reanalyzed (lines 15-28). Is the queue is empty, the algorithm stops (lines 30-31).

Lines 12 and 25, not only change the assignment of the replica objects to sites. The site's space and replica energies is considered. In line 12, the available storage space is increased for the site which moved the replica object, and reduced for the site that receives it with $size(F_i)$. Also, for the moved replica object and the replica objects connected to it the energy is reset. Similarly, in line 25, the available space of the site receiving the replica decreases and the energies of the affected replica objects is reset.

In Chapter 4 we present how we chose the move and replication thresholds.

Algorithm 2 GetBubbleEnergy

Input: *bubble*

1: W_{bubble} = this.BubbleEnergy {The current energy of each bubble is known}
2: $\mathcal{C} = \emptyset$ {set $\mathcal{C}$ keeps the replica objects that have the energy reset}
3: {Remove the changed energies and get the reset replica objects}
4: **for all** $R_k(F_i) \in bubble$ **do**
5: **if** $R_k(F_i) < 0$ **then**
6: W_{bubble} += $W_{R_k(F_i)}$
7: $W_{R_k(F_i)} = 0$
8: $\mathcal{C} = \mathcal{C} \cup \{R_k(F_i)\})$
9: **end if**
10: **end for**
11: {Compute the energies from the query springs}
12: **for all** $R_k(F_i) \in \mathcal{C}$ **do**
13: **for all** $Q \in \mathcal{Q}_{R_k(F_i)}$ **do**
14: W_{bubble} += $\mathcal{T}^Q_{quer}(R_k(F_i), Q)$
15: $W_{R_k(F_i)}$ += $\mathcal{T}^Q_{quer}(R_k(F_i), Q)$
16: **end for**
17: **end for**
18: {Add the energies from the correlation springs}
19: **for all** $\{R_k(F_i), R_l(F_j)\}$, $R_k(F_i) \in \mathcal{C}$, $R_l(F_j) \in \mathcal{C}$, $i \neq j$ **do**
20: **for all** $Q \in \mathcal{Q}_{R_k(F_i)} \cap \mathcal{Q}_{R_l(F_j)}$ **do**
21: W_{bubble} += $\mathcal{T}^Q_{corr}(R_k(F_i), R_l(F_j))$
22: $W_{R_k(F_i)}$ += $\mathcal{T}^Q_{corr}(R_k(F_i), R_l(F_j))$ {Assuming direction $R_k(F_i) \rightarrow R_l(F_j)$}
23: **end for**
24: **end for**
25: {Add the energies from the replication springs}
26: **for all** $\{R_k(F_i), R_l(F_j)\}$, $R_k(F_i) \in \mathcal{C}$, $R_l(F_j) \in \mathcal{C}$, $i = j$ **do**
27: **for all** $Q \in \mathcal{Q}_{R_k(F_i)} \cap \mathcal{Q}_{R_l(F_j)}$ **do**
28: W_{bubble} += $\mathcal{T}^Q_{repl}(R_k(F_i), R_l(F_j))$
29: $W_{R_k(F_i)}$ += $\mathcal{T}^Q_{corr}(R_k(F_i), R_l(F_j))$ {Assuming direction $R_k(F_i) \rightarrow R_l(F_j)$}
30: **end for**
31: **end for**
32: this.BubbleEnergy= W_{bubble}

Output: W_{bubble}

Algorithm 3 CheckMove

Input: $R_k(F_i)$, W_{init}

1: $S_{move} = site(R_k(F_i))$ {Return the original site if no better site found}
2: $W_{new} = W_{init}$
3: $\mathcal{S} = GetClosestSites(x, site(R_k(F_i)), size(F_i))$
4: **for all** $S \in \{\mathcal{S}\}$ **do**
5: $SimulateMove(R_k(F_i), S)$
6: $W_{virt} = GetBubbleEnergy(Bubble(R_k(F_i)))$
7: **if** $W_{new} > W_{virt}$ **then**
8: $S_{move} = S$
9: $W_{new} = W_{virt}$
10: **end if**
11: **end for**

Output: W_{new}, S_{move}

Algorithm 4 CheckReplicate

Input: $R_k(F_i)$, W_{init}

1: $S_{copy} = site(R_k(F_i))$ {Return the original site for the copy if no other site found}
2: $W_{new} = W_{init}$
3: $\mathcal{S} = GetClosestSites(x, site(R_k(F_i)), size(F_i))$
4: **for all** $S \in \mathcal{S}$ **do**
5: $SimulateReplicate(R_k(F_i), S)$
6: $W_{virt} = GetBubbleEnergy(Bubble(R_k(F_i)))$
7: **if** $W_{new} > W_{virt}$ **then**
8: $S_{copy} = S$
9: $W_{new} = W_{virt}$
10: **end if**
11: **end for**

Output: W_{new}, S_{copy}

Algorithm 5 SpringAlgorithm

Input: $(\mathcal{G}, \mathcal{P})$, $\mathcal{L}$

1: $R_k(F_i)$ = Extract($\mathcal{L}$)
2: $W_{now} = GetBubbleEnergy(Bubble(R_k(F_i)))$
3: **if** F_i.AllowsMovement **then**
4: $\{W_M, S\} = CheckMove(R_k(F_i), W_{now})$
5: **else**
6: $W_M = \infty$
7: **end if**
8: { σ is the move threshold }
9: $\sigma_M = AC(site(R_k(F_i)), S) \cdot size(F_i)$
10: {Move replica}
11: **if** $W_{now} - \sigma_M > W_M$ **then**
12: change $\mathcal{P}$ to place $R_k(F_i)$ on site S
13: add($\mathcal{L}$,$R_k(F_i)$)
14: **else**
15: **if** F_i.AllowsReplication **then**
16: $\{W_R, S\} = CheckReplicate(R_k(F_i), W_{now})$
17: **else**
18: $W_R = \infty$
19: **end if**
20: { σ_R is the replication threshold }
21: $\sigma_R = AC(site(R_k(F_i)), S) \cdot size(F_i) \cdot F_i$.nrReplicas
22: {Replicate}
23: **if** $W_{now} - \sigma_R > W_R$ **then**
24: change $\mathcal{G}$ to contain a new replica $R_l(F_i)$ of F_i
25: change $\mathcal{P}$ to assign $R_l(F_i)$ to site S
26: add($\mathcal{L}$, $R_k(F_i)$)
27: add($\mathcal{L}$, $R_l(F_i)$)
28: **end if**
29: **end if**
30: **if** $\mathcal{L} \neq \emptyset$ **then**
31: $\mathcal{G}$ = SpringAlgorithm($(\mathcal{G}, \mathcal{P})$, $\mathcal{L}$)
32: **end if**

Output: $(\mathcal{G}, \mathcal{P})$

Chapter 4

Experiments

This chapter presents the experimental results of the simulation model that was implemented to analyze the performance of the Spring Algorithm. There are two sets of experiments: first, we validate the performance of the Spring Algorithm by comparing it with an limited exhaustive search algorithm[1]; second, we verify the system's scalability by varying several parameters.

4.1 Experimental Setup

The simulator has three components: the *data generator module*, the *data cleaning module*, and the *Spring Algorithm simulator module*. The data generator modules creates a set of fragments, a set of sites, a set of queries, an initial random allocation of replica and query objects to sites – the pair $(\mathcal{G}, \mathcal{P})$. The queries are those that generate relationships between replicas and are representative of the access pattern to these replicas. We assume that, when a query accesses a fragment, it accesses the closest copy of that fragment. However, the queries that are randomly generated by this module do not consider this aspect and can access any replica of any fragment, which means that they do not simulate a real data access pattern. Therefore, a data cleaning module is used to rearrange the query springs so that they look more like real data access patterns. The Spring Algorithm simulator module implements the Spring Algorithm.

Details of each of these modules are given in the following sections. Here we only show how these modules interact (Figure 4.1). The data from the generator ($\mathcal{G}_0$, $\mathcal{P}$) are cleaned ($\mathcal{G}$, $\mathcal{P}$) and fed into the Spring Algorithm which returns a stabilized configuration ($\mathcal{G}_s$,

[1]Due to running time issues, we limit the exhaustive search space to the states where there are at most three replicas per fragment.

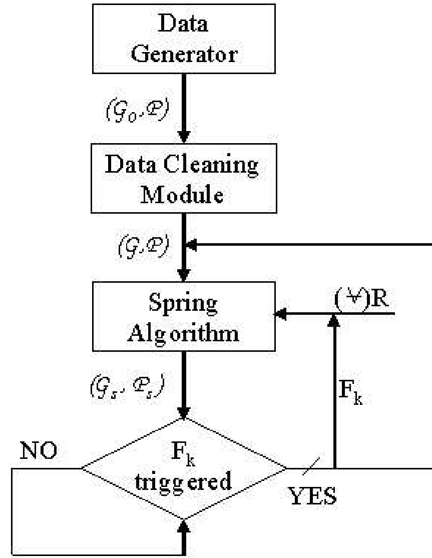

Figure 4.1: Module interconnection

$\mathcal{P}_s$). If a replica object $R_k(F_i)$ (could be a set of replicas) changed its access pattern, it is introduced into the queue $\mathcal{L}$. The algorithm analyzes the allocation configuration and adapts to changes. If the change in the access pattern to a replica is significant, the allocation configuration is changed, otherwise it is kept identical. The queue is similar to a switch for the Spring Algorithm module: when is not empty the module is on. The initial random fragment allocation is stabilized by analyzing all the replica objects (placing them in the queue). In Figure 4.1, this is shown as $\forall R$.

As mentioned when *CheckMove* was described, each replica object is analyzed based on the fixed allocation of the other objects in the bubble. This means that, at some point, an object could have been considered well allocated and eliminated from the queue based on a bad configuration that changed later when other replica objects in the queue were analyzed. To get a good distribution, we run the Spring Algorithm multiple times using all replicas as input. When an iteration of the algorithm stops, all objects are reintroduced into the queue and the algorithm is re-run. The reallocation stops when the bubble energy obtained in the nth iteration has not sufficiently changed. We consider that a good distribution is the one for which the bubble energy has decreased less than 1% in comparison with the one obtained in the previous iteration[2].

[2] In the current implementation, we do not cap the number of iterations, or the running time of the algorithm.

4.1.1 Data Generator Module

Given a set of parameters, the data generator creates some fictive data used to test the Spring Algorithm. Due to the nature of the bubbles, each bubble can be analyzed independently of the other. Since describing a single bubble suffices to characterize the environment, everything below refers to a single bubble.

The Sites

The **sites** are created given the total number of sites as a parameter. The available space on each site is a random value between two thresholds, CAP_{Min} and CAP_{Max}, which are set to 0MB and 10MB respectively. This simulates both sites with small capacity which are only used to run queries, and sites where the storage space is large and can store data. The available space is assumed to be distributed uniformly:

$$SITE.availSpace = \mathit{Uniform}(CAP_{Min}, CAP_{Max})$$

We only test how the number of sites, not how the sites' space availability, affects the system's performance. The default number of sites is 200.

The Fragments

Since the fragments in different bubbles do not interact, the fragments we generate are within a bubble. The **fragments** are created based on the following parameters: the number of fragments, the size of each fragment, whether or not the fragment allows movement and replication, and the initial number of copies.

- We want to see how increasing the number of fragments affects the system, therefore, in one test, we vary the number of fragments. The default value is 200.
- The size is generated with $\mathit{Uniform}(\mathit{SIZE}_{Min}, \mathit{SIZE}_{Max})$, similar to the available space on the sites. We choose the value of SIZE_{Min} to be 1KB and SIZE_{Max} to be 250KB.
- Some fragments allow replication, others don't. The percentage of replicating fragments is given by a parameter $REPLICATE_{Yes/No} \in [0, 1]$, that we choose to be 0.6. If a fragment does not allow replication it has only one replica object. Similarly, the percentage of fragments that allow movement is $MOVE_{Yes/No} \in [0, 1]$, and we choose it to be 0.8. As mentioned in Chapter 3, *allow movement* and *allow replication* apply to all replicas of a fragment.

- The fragments that allow replication can initially have a randomly generated number of replicas. This is generated with $Uniform(REPLICAS_{Min}, REPLICAS_{Max})$, where $REPLICAS_{Min}$ is 1 and $REPLICAS_{Max}$ is 4. After creation, each of these replicas is allocated to randomly picked sites.

The Queries

The **queries** are created using the following parameters: the number of queries, the update-to-retrieval ratio, the selection-to-join ratio, the data access distribution, the query frequency, and the spring characteristics. The values for the first four parameters are varied throughout the experiments and will be presented later.

- The *number of queries* defines the size of the set of queries that characterize the fragments' access patterns. The default value is 5000.
- The *update-to-retrieval ratio*, $QUERY_{U/R} \in [0, 1]$, defines the proportion between the number of updates and the number of reads. It is used when creating the update/read queries. In the default setting we have 80% reads and 20% updates.
- Within the set of read queries, the *selection-to-join ratio*, $QUERY_{S/J} \in [0, 1]$, defines the ratio between selection queries (those that have one spring) and join queries (those formed by many springs). In the default setting we choose to have 50% selections, 50% joins.
- The *data access distribution* defines the query distribution to fragments. Two sets of queries are generated based on the same site setting and fragment allocation. The first set represents a uniform allocation, therefore each fragment has equal probability of being accessed by a query. The second set represents an 80/20 distribution in which 80 percent of the queries access 20 percent of the data, simulating hot spot accesses. The default distribution is 80/20.
- The *frequency* of the query represents the number of times the query is executed. We chose this value to be fixed to 1 throughout the experiments, in order to generate unique queries.
- The *spring characteristics* define the attributes of the springs that comprise the query, and are fixed throughout the experiment.

    1. *The size of the transferred data associated with the read query springs* is defined with
$$Uniform(DATASIZE_{ReadMin}, DATASIZE_{ReadMax}),$$

where the first is 1KB and the second is 40KB.

2. *The number of springs that comprise the join queries* is defined with

$$Uniform(NRSPRINGS_{Min}, NRSPRINGS_{Max}),$$

where the parameters have values 1 and 4. It represents the number of springs that are created to simulate the execution of a join.

3. *The size of the transferred data associated with the correlation springs* is defined with

$$Uniform(DATASIZE_{CorrMin}, DATASIZE_{CorrMax}),$$

where the parameters have the same values 1KB and 40KB.

4. *The size of the data associated with the update query spring* is the same as the one associated to any of the replication springs in the same query[3]. The size is defined by $DATASIZE_{UpdMaxPer}$, which represents the maximum percentage of the fragment that is updated, and we set it to 0.15. To simplify the implementation of the Spring Algorithm and the limited exhaustive search algorithm, in our experiments the update query is composed of a query spring connecting the query object and the updated replica object, and of other springs connecting the updated object to each other replica object.

We want to find, in average, how many replicas-per-fragment are created. There are 40% of the fragments that do not allow replication, and 60% that do. From those that do, in average, the number of replicas is 2.5 (minimum 1, maximum 4). Therefore, there are $0.4 \times 1 + 0.6 \times 2.5 = 1.9$ replicas-per-fragment. Regarding the number of springs, in the default setting there are 40% selections which have one spring, 40% joins which have on average 3 springs (minimum 2, maximum 4), and 20% updates which connect on average two objects (1.9 replicas/fragment). That means, in average, there are $1 \times 0.4 + 3 \times 0.4 + 2 \times 0.2 = 2$ springs for query.

The data set that was created with these default values creates highly replicable hot spots as 80% of the queries access only 20% of data and 80% are read operations. Also, the way we chose the size of the transferred data associated with query and correlation springs also affected the data set. We assigned a value between 1 and 40KB, and did not consider the size of the fragment actually involved in the query. Therefore, a fragment with size 1KB can be connected with query springs that transfer 40KB. The gain from local

[3] A query that performs an update is represented by a query spring and one or more replication springs.

read is high (40KB), and the cost to keep the data updated (0.15×1KB) is low, therefore the fragment would be highly replicable. However, this is an extreme case. We chose the value of σ_M to be the cost of moving the object from the source site to the destination site. For replication, σ_R was chosen as the cost to move the replica times the number of existing replicas of that fragments (to bound the replication). The way the experiments were influenced by σ_R is detailed in the scalability tests.

4.1.2 Data Cleaning Module

As previously mentioned, this module rearranges the springs that were randomly created by the data generator. We want to simulate read accesses, therefore the springs should be reconnected to use closer replicas. The cleaning process is given in Algorithm 6. For every query and then for every fragment accessed by that query, this algorithm checks if there is a closer replica that can be used. This is a heuristic algorithm that gives a reasonable configuration. An algorithm that would find the best query configuration would have, for each query accessing N fragments, N nested loops (each with number of replica steps), whereas algorithm 6 performs N loops, one after another.

Algorithm 6 Rearrange the springs

1: **for all** Q, $Q \in \mathcal{Q}$ **do**
2:   **for all** F, F accessed by Q **do**
3:     $R_{best} = R_k(F)$, $R_k(F)$ replica currently accessed by Q
4:     $C_{best} = Cost(Q, R_{best})$, query execution cost if R_{best} is used
5:     **for all** R, $R \in \{R_i(F)\} - R_k(F)$, $i = 1, .., M$ **do**
6:       **if** $Cost(Q, R) < C_{best}$ **then**
7:         $C_{best} = Cost(Q, R)$
8:         $R_{best} = R$
9:       **end if**
10:     **end for**
11:     **if** $R_{best} \neq R_k(F)$ **then**
12:       Connect Q to R_{best}
13:     **end if**
14:   **end for**
15: **end for**

When re-arranging a query, the algorithm first sums up the energies of all the springs in the query (line 4). The result represents the cost of executing the query. Next, to improve this cost, for each fragment involved in the query, all of its replicas are analyzed. If a lower

cost is found using a different replica object, then the springs in the query are connected to that replica (line 12).

Running the data cleaning module is not necessarily required as the Spring Algorithm does the spring re-configuration anyway. However, using data that's not cleaned increases the running time of the algorithm – and there is no significance in that.

4.1.3 Spring Algorithm Simulator Module

As mentioned, the Spring Algorithm Simulator takes as input a $(\mathcal{G}, \mathcal{P})$, and a replica or set of replica objects that changed their access pattern. It reallocates the replicas such that the given set of queries is performed more efficiently. The query set $(\mathcal{Q})$ changes gradually since the last reallocation, therefore the set of queries should make minor changes to the allocation configuration.

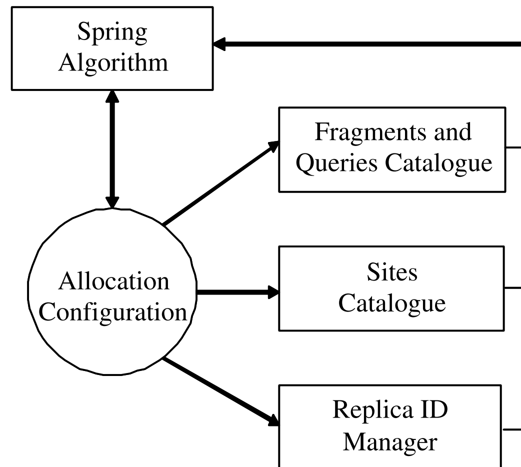

Figure 4.2: The Spring Algorithm Simulator

The Spring Algorithm simulator uses three components: the Fragments and Queries Catalog, the Sites Catalog, and the Replica Id Manager (Figure 4.2).

- The **Fragments and Queries Catalog** keeps track of where the replica objects and query object are located by implementing $site(R_k(F_i))$ and $site(Q)$. Another function of the catalog is to simulate that a replica object is reallocated on a different site in order to verify whether or not this is a good decision (*SimulateMove* and *SimulateReplicate*). It also checks if a replica object of a fragment already exists at a specified site, used when removing a replica.

- The **Sites Catalog** contains information about the average delay-per-kilobyte when sending data between sites ($AC(S_i, S_j)$). This module also gives information about sites located in the proximity of another site ($GetClosestSites(x, S, size(F))$). To simplify the implementation, this method returns the x closest sites that are willing to store the replica object.

- Each fragment and each replica of that fragment have their own unique id. The ids are monotonically increasing integer values. The **Replica Id Manager** contains the global information about the replica ids in the system. If a replica is removed, its id is erased from the list of those in use, and reused later when a new replica is created.

4.1.4 Experimentation Platform

The sites, the fragments, and the queries are three factors we tested to see how they influence the system performance. In Section 4.1.1, we described the parameters that were used when generating them. We test the system to see how it reacts when related parameters change.

The **number of sites-per-bubble**, S, is a factor that needs to be considered. We want to see if the system is scalable, therefore this parameter is varied to verify the system's response time. S is set to 50, 100, 200, and 1000.

We want to see if the system is scalable when the **number of fragments-per-bubble**, F, is increased. We use 100, 200, 500, and 1000 as test values.

The **number of queries-per-bubble**, Q, influences the response time since the time to compute the energy of a replica object depends on the number of springs connected to that replica object. Also, since each time the replica object is moved, replicated, deleted, or simulated for replication the springs are rearranged, an increasing number of queries would increase the running time. We want to see whether our system scales well with increased number of queries accessing the fragments in the bubble. We use 2000, 5000, 10000, and 50000 as test values.

Besides their number, the types of queries are also important. The **update-to-retrieval ratio**, U/R, of queries can be one factor. If the number of updates is high, the number of replicas of each fragment should decrease as the overhead of maintaining these replicas will be high. As a result, it should take less time to stabilize the system when there are more updates than when there are fewer. We test the system for 10%, 20%, and 30% updates.

Selection-to-join ratio, S/J, of queries is another factor. More selections should reduce the time to stabilize the spring system for two reasons. First, there are fewer springs in the system, therefore the time to compute the energies of replica objects is reduced. When the reallocation is performed, there are many simulations that reset the energy of

replica objects forcing the re-computation by summing up springs energies. Second, more selections mean that replica objects are not as tightly connected as in the case of joins that create correlations between replicas. Therefore, reallocating a replica object will not cause other replica objects to reallocate, and this results in less time to stabilize the system. In our experiments we vary the selection-to-join ratio so that we have 25%, 50% and 75% selections.

In real environments, the **data access distribution**, D, of queries corresponds to the 80/20 distribution, that is 80 percent of the queries access 20 percent of the data. We want to see the response time for an 80/20 versus an uniform distribution.

Only these six parameters are varied. When one of these parameters is varied, the others are set to their default values (Table 4.1).

Table 4.1: The Default Values in the Experiments

Parameter	Explanation	Default value
S	No. of sites / bubble	200
F	No. of fragments / bubble	200
Q	No. of queries / bubble	5000
U/R	Update-to-retrieval ratio of queries	1/4
S/J	Selection-to-join ratio of queries	1/1
DAD	Data access distribution of queries	80/20
SmC	Site minimum capacity	0MB
SMC	Site maximum capacity	10MB
mSF	Minimum size of fragments	1KB
MSF	Maximum size of fragments	250KB
AR	Percentage of fragments allowing replication	60%
AM	Percentage of fragments allowing movement	80%
ImR	Initial minimum number of replicas	1
IMR	Initial maximum number of replicas	4
ϕ	Query frequency	1
QmS	Query spring minimum data size	1KB
QMS	Query spring maximum data size	40KB
CmS	Correlation spring minimum data size	1KB
CMS	Correlation spring maximum data size	40KB
mSJ	The minimum number of springs per join	1
MSJ	The maximum number of springs per join	4
MUP	Maximum update percentage	15%

4.2 Experiments and Results

We performed two classes of experiments:

- The first set of experiments validates the simulator by comparing the results with a limited exhaustive search algorithm.
- The second set of experiments involve scalability tests with respect to the parameters discussed above (performance evaluation experiments).

In the performance experiments, the primary metric is the time to complete the reallocation. Secondary metrics are: the number of replicas in the system; the minimum, the maximum, the mean, and the median number of replicas-per-fragment; the number of times the objects were moved and the number of times they were tested for moving but were not moved; the number of times the replica objects replicated and the number of times they were tested for replication but did not replicate; and the number of times the replica objects merged. All these are interpreted when analyzing the test results.

All of the experiments are run on a 2.2MHz Pentium 4 machine with 512MB RIMM memory. Since the IOs are performed only at the beginning of the tests, the hard drive speed is not relevant, but only the processor, the memory speed, and the size of the memory. The program was written in Java.

4.2.1 Validation Experiments

As mentioned, to verify the system's performance, we choose to compare the Spring Algorithm with a limited exhaustive search. Because the running time increases explosively we limited the maximum number of replicas that any fragment has to three. For instance, even the limited time algorithm ran for almost 8 hours for a simple test set (Table 4.2).

The goal of this set of tests is to find out how good the heuristic used in the Spring Algorithm is. The three parameters changed are the number of sites, the number of fragments in the system, and the number of fragments that allow replication. Therefore, three sets of tests are performed.

The results of our test are presented in Table 4.2. For the limited exhaustive search algorithm we also show the number of cases that have been analyzed.

In each of the analyzed cases there were 1000 queries generated to access the fragments. As mentioned, in average, each query has two springs. The objects were hooked with about 2000 springs. The default values for the number of fragments, sites, and replicating fragments are 5, 5, and 2, respectively. For easy characterization, we call this test "552".

Table 4.2: The Validation Experiments

Parameters			Limited exhaustive algorithm			Spring Algorithm		Ratio
Fr	*Sites*	*Rep*	Time	Access Cost	Cases	Time	Access Cost	
5	4	2	0:16:09	264,766.06	15,625	00:00:04	336,804.16	0.79
5	5	2	1:07:13	272,195.12	78,125	00:00:03	272,195.12	1
5	6	2	4:51:58	318,158.88	390,625	00:00:03	332,999.84	0.95
4	5	2	0:15:23	228,579.88	15,625	00:00:05	284,305.66	0.80
5	5	2	1:07:13	272,195.12	78,125	00:00:03	272,195.12	1
6	5	2	5:18:54	193,206.16	390,625	00:00:04	188,346.80	1.02
5	5	0	0:01:53	325,421.88	3,125	00:00:01	325,421.88	1
5	5	1	0:11:24	298,938.53	15,625	00:00:04	487,775.97	0.61
5	5	2	1:07:13	272,195.12	78,125	00:00:03	272,195.12	1
5	5	3	7:41:09	233,902.75	390,625	00:00:04	243,289.72	0.97

The default test is performed only once, even though (for comparison reasons) it is shown three times in Table 4.2.

As previously mentioned, the energy of the bubble represents the sum of the energies of replicas within the bubble. In the analyzed cases, because there are few replicas, each replica object has associated with it a significant amount of the total energy. As a result, each misallocation would produce important changes in the bubble energy. However, in a large system where there are many objects, placing an object at the optimal site or at a site next to it would represent only a minor difference in the bubble energy.

We notice the following: (a) in four tests the Spring Algorithm gave very good results, as measured by the closeness of the cumulative access cost of the queries in the reallocated system, (b) one test returned a result even better than the one returned by the limited exhaustive search algorithm, and (c) in three tests the results were far from optimal. All of these are presented next, but we focus more on those that returned bad results and analyze why that happened.

Tests 552, 562, 550, and 553 give very good results. The difference in the energies is due to suboptimal spring connections. The example presented in Figure 4.3 shows that it is possible to have a query which is not connected optimally and yet it is not detected by the rearrange algorithm (Algorithm 6). In the initial allocation presented in Figure 4.3(a), the energy of the query is 6 ms. Figure 4.3(d) presents a case in which the spring connection is optimal, but the rearranging algorithm fails to consider it because it only analyzes a replica at a time. The arranging algorithm may try to use the second replica of

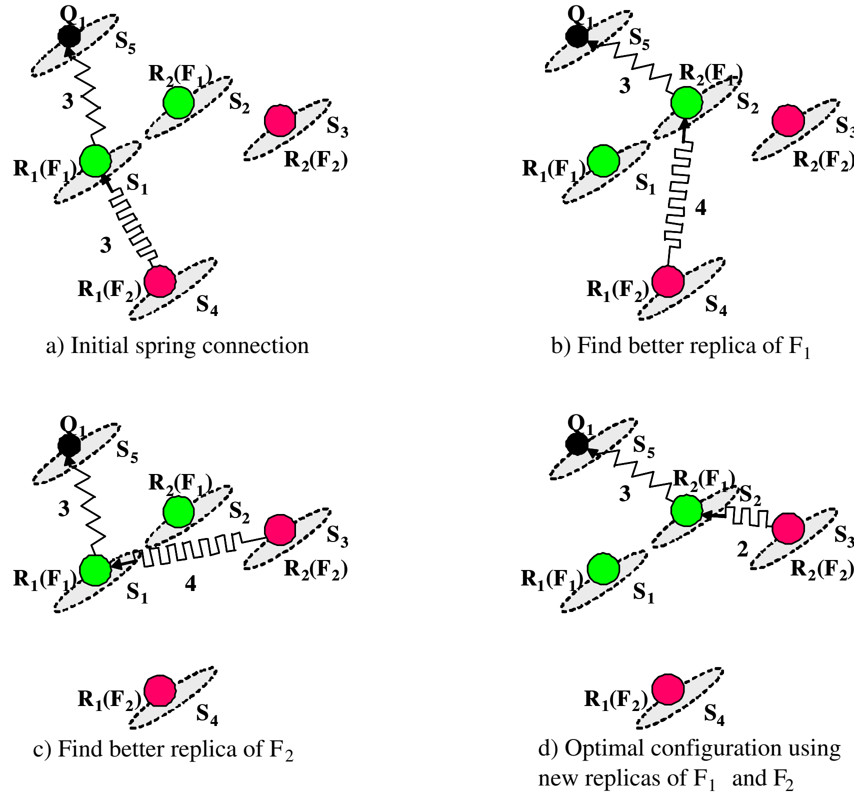

Figure 4.3: The heuristic rearranging algorithm analyzes one fragment at a time.

F_1, but does not find a lower energy (Figure 4.3(b)). Similarly, in Figure 4.3(c), using the other replica of F_2 does not return a lower energy.

In test 550, since there is only one copy for each fragment, the returned result is identical to the one returned by the limited exhaustive algorithm. However, in the other three cases, even though the allocation configuration is the same, the energy that the heuristic Spring Algorithm returns is higher due to the improper spring connections mentioned above. We believe that in a real environment this is not likely to be an issue, since a query optimizer will use the new allocation when optimizing the execution of new queries.

Test 652 returned a result, better than the one returned by the limited exhaustive search algorithm. This is simply due to the fact that the Spring Algorithm placed one of the two replicable fragments on all five sites, whereas the exhaustive search algorithm was constrained to analyze up to three replica objects of each fragment. As a result, it failed to analyze the allocations with four replicas or more.

For three tests – 542, 452, and 551 – the best allocation our algorithm found did not return a good score. The reasons why it did not perform so well (other than suboptimal connection of the springs) are described next.

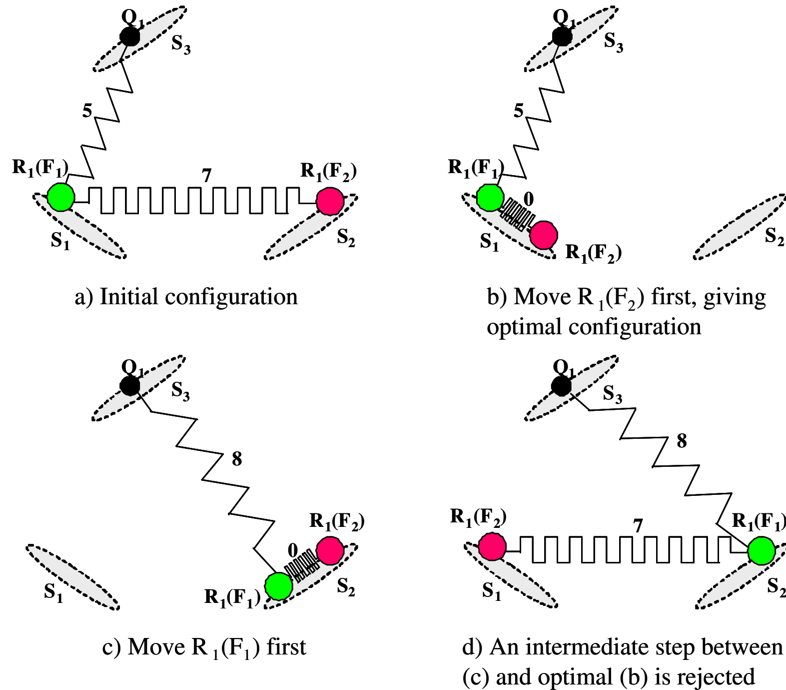

Figure 4.4: Replica objects' order of analysis is important.

The first issue is **the order in which the replica objects are analyzed**. For the join in Figure 4.4, let us consider the size of the data transferred associated with each spring to be 1KB so that numerically speaking the energy of the springs reflects the access cost between sites. The average access cost between S_1 and S_2 is 7ms/KB, between S_1 and S_3 is 5ms/KB, and between S_2 and S_3 is 8ms/KB.

Figures 4.4(b) and (c) illustrate how two configurations can be obtained due to different order of analysis of fragments. In Figure 4.4(b) we analyze $R_1(F_2)$ first and we find the optimal configuration. If $R_1(F_1)$ is moved first (Figure 4.4(c)), the energy in the new configuration is less than in the original configuration, $8ms < 12ms$, therefore the algorithm moves $R_1(F_1)$. However, when $R_1(F_2)$ is analyzed and virtually placed on S_1 (Figure 4.4(d)), the configuration is discarded as being too expensive, even though it is an intermediate step to the optimal configuration. As a result, the Spring Algorithm uses Figure 4.4(c) as the new allocation even though it is suboptimal.

The allocations presented in Figure 4.4(b) and (c) produce different results from the same initial configuration. Figure 4.4(c) is a case of misallocation that the algorithm will not detect. This demonstrates the importance of finding the best order to analyze the replica objects; this is not an issue we address in this thesis.

We argue that, in a large environment, the order in which the replica objects are analyzed is not as big an issue as in a small environment. The misallocation occurs when the sites are neighbors, therefore having both replica objects on S_1 or on S_2 does not influence (in percentage) the bubble energy as much as in a small setting. Our algorithm may not place the objects on the optimal sites, but it keeps them within a reasonable range of the optimal allocation.

The second issue is that **the algorithm does not reconnect the springs when simulating a move**, but only when simulating a replication.

Figure 4.5(a) shows an example in which the initial allocation of three replicas (where the energy is 11ms) is changed to a configuration in which the energy is 9ms (Figure 4.5(c)). Even though site S_2 is considered as a potential candidate, Figure 4.5(b), the fact that Q_2 did not connect to $R_1(F_1)$, which is now closest and where the energy of the spring is only 3ms, made the difference: S_2 is not a good site to store $R_3(F_1)$. Looking at Figure 4.5(b) and (c) we notice the same allocation of the objects. However, the algorithm fails to detect that. In Figure 4.5(c) the two replicas are placed on S_2 to make the figure intuitive.

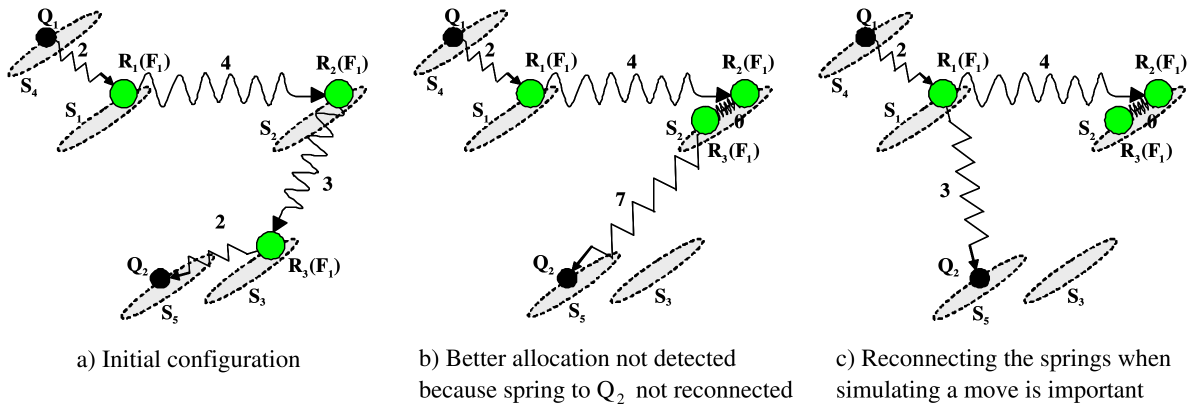

Figure 4.5: In a small setting it is crucial to reconnect the springs for each simulation.

For each of the three cases that returned bad results, the initial allocation was created with more than the optimal number of replicas. The algorithm failed to merge replicas, hence the result. Because the energy of each replica represents a large percentage of the bubble energy, this energy is significantly higher than optimal. As mentioned, $R_3(F_1)$ is dropped before being moved to a site that already has a copy.

Reconnecting the springs for each move simulation is vital as the environment is small and each replica has a large percentage of the bubble energy (final score). However, for large environments this is not feasible, as for each replica object there are many simulations to move it to various neighbor sites, and there could also be many springs connected to it.

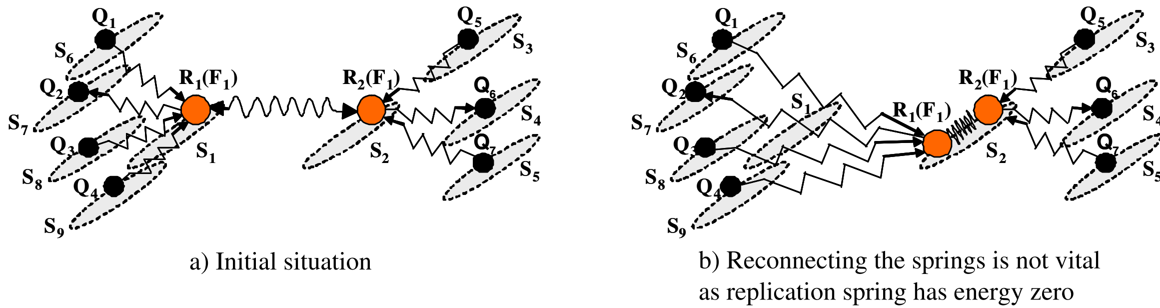

Figure 4.6: In large environments spring reconnection is not important.

We argue that in large environments, not reconnecting the springs when simulating moves is not an issue, as illustrated by Figure 4.6. The object $R_1(F_1)$ is simulated to be at S_2, which is the good allocation. The springs are still connected to each individual replica. In this case the allocation is good, since the energy of the replication spring between the two replicas is zero (the replicas are located at the same site) and all the springs are connected to a replica located at site S_2. However, for small test cases, reconnecting the springs makes a lot of difference.

We retested the three cases in which we got poor results and reconfigured the springs for each move simulation. We ran the Data Cleaning Module after the object was virtually placed at other sites. When the simulation ended, we ran the module again in order to return to the previous spring configuration. The returned results were very good, but the running time increased significantly. For case 542 the bubble energy was $267,421.28$ and the running time was approximately one minute and twenty seconds. For 452 the result was $260,101.98$ and the running time about one minute and forty seconds. For 551 the result was $298,938.44$ and the running time about one minute and twenty-five seconds. The difference between these results and the ones returned by the limited exhaustive search algorithm is probably due to rounding issues.

We have seen that rearranging the springs for each move simulation gives much better results. However, the running time increases significantly because rearranging the springs (in this simple case 2000) is the most time consuming process. For example, when moving an object to a neighbor site, first the x closest sites are analyzed, then the object is moved

to that site. In this situation the springs are rearranged $x + 1$ times, whereas in our approach the springs are rearranged only after the object has been moved. The ratio is $(x + 1)$-to-1.

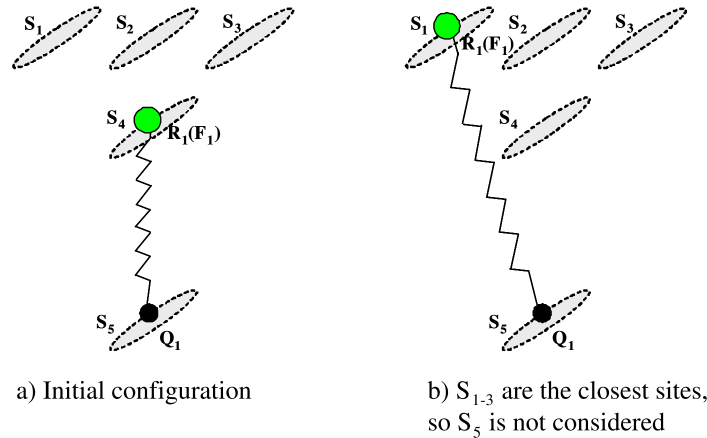

Figure 4.7: The number of analyzed sites is important.

The third issue that could influence the results is **the number of neighbor sites that are analyzed**. In Figure 4.7(a) a query object is connected to a replica object. Let us consider that the site at which the query object is placed would accommodate $R_1(F_1)$. However, the object will never be allocated to the best location where the access cost is zero because only the three closest sites S_1, S_2, and S_3 are analyzed. As shown in Figure 4.7(b), the simulation always finds a worse cost when moving the object, therefore the object is kept at that site, even though this is not optimal. However, the number of sites that are analyzed does not influence the results of the test because our algorithm considers all the sites for simulation. However, we noticed in the performance tests that are presented next, that this number is relevant. In a geographical setting for instance, the number of sites should be the minimum number of sites that cover all surrounding regions. A smaller number of analyzed sites makes the Spring Algorithm faster, as fewer simulations are performed each time the replica objects are simulated for move or replication. However, these simulations return worse results as they fail to analyze all relevant candidate sites. In the spring system, a lower number of analyzed neighbor sites would translate into a barrier that does not allow the object to move in a direction other than the direction of the neighbor sites.

We consider the verification results to be encouraging. We discovered the weaknesses of our algorithm, but we believe that in the Internet environment, which is much larger, these

would not be major issues. In the second part of this chapter we present the performance of the Spring Algorithm and show how the system scales to larger environments.

4.2.2 Performance Evaluation Experiments

This section analyzes the system response in larger environments, by varying related parameters. The goal is to check if the system is scalable and if the running time increases exponentially when the values of the parameters are increased linearly.

Since the parameters were presented in Section 4.1.4 we only enumerate them here:

- number of sites-per-bubble
- number of fragments-per-bubble
- number of queries-per-bubble
- update-to-read ratio
- selection-to-join ratio
- data access distribution

We test one of these parameters at a time. The other parameters are set to their default values as shown at page 53 in Table 4.1.

Scalability with Respect to the Number of Sites in the Bubble

We tested the system to see whether it responds well when the *number of sites-per-bubble* increases. We chose to test the system with 50, 100, 200, and 1000 sites. The first four test sets are generated randomly to see how the algorithm reacts in randomly generated network topologies. In order to simulate a real network topology, due to the lack of publicly available information, we used 200 node stations of the Romanian Railway System and the distance (in kilometers) between them, and approximated the distance as the cost of transferring 1KB of data, in milliseconds.

Because the notion of locality is better defined in the realistic setting, the replica objects move more than in the generated one. For the same reason, in the generated setting the fragments replicate more because the gain of having data locally is high. In Figure 4.8, two replicas are needed to serve the queries efficiently. Sites S_1 and S_3 are not in the same vicinity, even though the access cost (the dotted lines) between S_1 and S_2 and between S_2

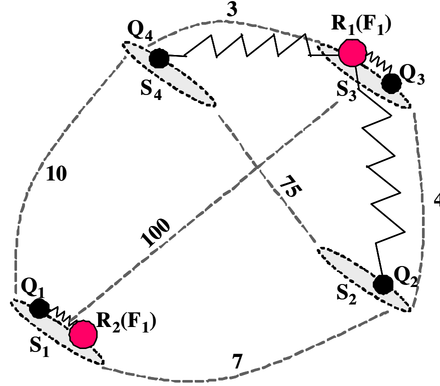

Figure 4.8: Randomly generated sites encourage replication.

and S_3 is low. The reason why they are not in the same vicinity is because the access cost is generated randomly. In a real site setting, the four sites would be in the same vicinity.

The number of replicas obtained after the reallocation has been performed is directly proportional to the number of sites (Table 4.3) because now more fragments are hot spots (20% of the total number of fragments are hot spots). As the number of replicas in the system increases, the time required to analyze them increases as well, as presented in the table.

When empirically choosing the number of neighbor sites to be analyzed, for the realistic set the performance increased dramatically from 1 to 5 analyzed neighbors, then slowly increased up to 10 neighbors, and stabilized afterwards. For the generated data set, the increase was constant and more neighbors returned better results even after the limit of 10.

All other tests use only the realistic network topology. Since we want to see how the algorithm reacts when different parameters change, the sites configuration is fix.

Scalability with Respect to the Number of Fragments in the Bubble

The running time is obviously influenced by the *number of fragments* because more fragments take more time to analyze. Each fragment can have more than one copy, therefore there may be more replica objects in the system than the actual number of fragments (we showed that in average there are 1.9 replicas-per-fragment).

In the experiments (Table 4.4), for 200 and 500 fragments, we obtained a shorter running time than for 100. This happens because of the following:

- In a small set of objects, since the queries access only few objects, each object is

Table 4.3: Performance Evaluation w.r.t. Nr. of Sites

	Test	1	2	3	4	5
	Nr.Sites	50	100	200	1000	200
Init. Conf.	Nr.Replicas	382	371	370	359	370
	Rpl/Frg (min)	1	1	1	1	1
	Rpl/Frg (max)	4	4	4	4	4
	Rpl/Frg(mean)	1.91	1.86	1.85	1.78	1.85
	Rpl/Frg(med)	1	1	1	1	1
Final Conf.	Nr.Replicas	480	475	504	621	403
	Rpl/Frg (min)	1	1	1	1	1
	Rpl/Frg (max)	20	13	20	20	9
	Rpl/Frg(mean)	2.4	2.38	2.52	3.11	2.02
	Rpl/Frg(med)	2	2	2	3	2
Oper.	Moved	96	116	139	184	375
	Not Move	2122	1686	1904	2380	1405
	Replicate	98	104	134	262	34
	Not Replicate	1704	1306	1510	1854	1111
	Merged	0	0	0	0	1
Bubble Energy	Initial	1.25E7	2.58E7	5.40E7	2.55E8	5.71E7
	Final	8.79E6	1.90E7	4.02E7	1.77E8	4.19E7
	Running time	5:45	3:11	6:41	9:59	5:35

connected with more springs. In a large set of objects, since the same number of queries access more objects, each object is connected with fewer springs. This means that in the first case computing the energy of an object takes more time as each object's energy is the sum of more springs.

- In the spring system, when an object is moved, the objects directly connected to it are affected. In a small set of objects, each object is connected to a larger percentage of the total number of objects. This means that computing the energy of the bubble takes more time.

Just to give an example, when 15 queries (30 springs) access three objects, each object is on average connected with 10 springs. When the same number of queries accesses 5 objects, on average, 6 springs are connected to each object. It is obvious that computing the energy of an object connected with 10 springs takes more time than for one connected with 6 springs. As the first setting is small, moving an object affects the other two, and

Table 4.4: Performance Evaluation w.r.t. Nr. of Fragments

	Test	1	2	3	4
	Nr.Fragments	100	200	500	1000
Init. Conf.	Nr.Replicas	199	358	991	1811
	Rpl/Frg (min)	1	1	1	1
	Rpl/Frg (max)	4	4	4	4
	Rpl/Frg(mean)	1.99	1.79	1.98	1.81
	Rpl/Frg(med)	1	1	1	1
Final Conf.	Nr.Replicas	221	385	1040	1890
	Rpl/Frg (min)	1	1	1	1
	Rpl/Frg (max)	7	5	7	17
	Rpl/Frg(mean)	2.21	1.93	2.08	1.89
	Rpl/Frg(med)	2	1	2	1
Oper.	Moved	238	312	557	796
	Not Move	725	1244	3400	6070
	Replicate	22	28	49	79
	Not Replicate	591	980	2783	4511
	Merged	0	1	0	0
Bubble Energy	Initial	4.74E7	5.05E7	5.08E7	5.05E7
	Final	3.64E7	3.92E7	4.05E7	4.11E7
	Running time	6:01	5:10	4:55	9:05

to compute the energy of the bubble all 30 springs have to be considered. As the second setting is larger, moving an object does not affect all other objects. In this case, to compute the energy of the bubble, less than 30 springs are considered. For simplicity, in this example we considered uniform distribution of queries to objects.

The combination of the two factors previously presented (the two bullets) influences the running time in opposite ways, as shown in Table 4.4. Also, notice that the number of replicas in the system increases, but not proportional with the number of fragments. In fact, the higher the number of fragments, the fewer replicas-per-fragments are created. There are fewer fragments accessed by the same number of queries, therefore there are more springs that pull from the fragments to make them replicate. In the first test, there are 22 new replicas per 100 fragments, whereas in the other cases there are 14, 10, and 8 new replicas created per 100 fragments. Similarly, on average, in the first test each replica in the system moves more than once, whereas in the other cases it moved less than once. The hot spot fragments may replicate and move, but the others are not accessed sufficiently to

justify the cost of moving them. For this reason, merging operations are not performed. However, just to test our hypothesis, we changed the algorithm to allow a fragment to move if there is a better cost, removing the threshold σ_M. As a result, cold replicas merged 69 times because initially the fragments had more replicas than was needed.

Scalability with Respect to the Number of Queries in the Bubble

In these tests we can see again the effect of the highly replicable data set, as the mean number of replicas/fragment increased from 1.79 to 2.46 for 50000 queries (Table 4.5). The hottest fragment has increasingly more replicas. However, the effect of σ_R and the way we chose it for replication is reflected in the results. The greater the number of replicas in the system, the higher the threshold that needs to be overcome in order to replicate the fragment further.

Table 4.5: Performance Evaluation w.r.t. Nr. of Queries

	Test	1	2	3	4
	Nr.Queries	2000	5000	10000	50000
Init. Conf.	Nr.Replicas	358	358	358	358
	Rpl/Frg (min)	1	1	1	1
	Rpl/Frg (max)	4	4	4	4
	Rpl/Frg(mean)	1.79	1.79	1.79	1.79
	Rpl/Frg(med)	1	1	1	1
Final Conf.	Nr.Replicas	377	392	423	491
	Rpl/Frg (min)	1	1	1	1
	Rpl/Frg (max)	11	14	24	30
	Rpl/Frg(mean)	1.89	1.96	2.12	2.46
	Rpl/Frg(med)	1	1	2	2
Oper.	Moved	163	310	501	838
	Not Move	1480	1260	2060	2506
	Replicate	19	35	65	134
	Not Replicate	1156	981	1629	2006
	Merged	0	1	0	1
Bubble Energy	Initial	2.00E7	4.99E7	1.01E8	5.05E8
	Final	1.69E7	3.95E7	7.71E7	3.70E8
	Running time	0:48	5:10	28:12	9:30:26

Regarding the number of moves, it is interesting to see that except for the first test,

the increase is logarithmic.

In the first test, there are 2000 queries which correspond to approximately 4000 springs. 800 (20%) of these springs are connected to 160 (80%) of the fragments. On average, each of the $0.8 \cdot 358 = 286$ cold objects are connected with approximately $800/286 = 3$ springs. Unless the data size associated with these queries is high to create of a strong spring, the cold objects will not move. This explains why there are fewer replica objects moved in the first test than in the second. In the first test, the objects move closer to the optimal site, but do not have enough energy to get in its neighborhood. As a result, the fragments do not move many steps. In the second test, increasing the number of queries generated more energy to move the replica objects. Due to the high number of queries that access replica objects allocated at non-optimal sites, the algorithm was able to detect that there was a high performance loss that overcome the cost σ_M of relocating the object. Therefore the object is placed more precisely with the number of queries. For 5000 queries there are 10000 springs in the system, which means, on average, that each cold object is connected with $2000/286 = 7$ springs. For 5000 queries the objects are already placed at good locations. Increasing the number of queries increases the precision, but the object may just move another step since it is already in the optimal neighborhood.

The increase in the number of moves between the second, third and fourth tests is explained, as mentioned, by the number of queries which result in more precise allocation. Another factor that influenced this increase is the number of new replicas created. The hot fragments became even hotter creating more replicas that moved and counted in the total number of moves.

When analyzing the obtained energies, it can be seen that the initial ones are directly proportional to the number of queries. However, the final energies are reduced to 85%, 80%, 77%, and 74% of the initial one, as showing again that more queries represent more energy to move the fragments towards the optimal location.

We noticed that each time the query number doubles, the running time increases significantly. Twice the time to compute would be normal, as there are twice as many springs. However, for more randomly generated queries, each replica object is directly connected to more objects. When simulating a move/replicate the energies of all these replica objects are reset, and as computing the energy of a replica objects from springs is expensive, this is reflected in the running time.

Scalability with Respect to the Selection-to-Join Ratio

The following three experiments are shown in Table 4.6. All these data sets were created at the same time, given as parameters the selection-to-join ratio, the update-to-retrieval ratio, and the data access distribution. Except for queries, all the obtained configurations are

identical. Furthermore, even the query allocation is the same. This means that the query objects are placed on the same sites in all configurations, even though they are actually different otherwise.

We changed the proportion of the selection-to-join ratio to 25/75, 50/50, and 75/25 to see how the system reacts to this parameter. We noticed that the maximum number of replicas-per-fragment decreases with the number of joins. Also, overall, there are fewer copies created when running the algorithm on the data set with many more selections. These two are the results of fewer correlation springs in the system. Remember that for a query doing a join $R_1(F_1) \bowtie R_1(F_2)$, such that $R_1(F_1)$ is connected with a spring to $R_1(F_2)$ and $R_1(F_2)$ is connected to the query object, the effect of the fixed query object is pulling both fragments ($R_1(F_1)$ indirectly) close to its location. Therefore, in the case of joins, the query objects pull more than just one object towards them. As a result, more correlation springs connected to objects create more pressure on the replica objects and make them replicate.

In the results table we see another effect of fewer joins in the system. Fewer joins mean there are fewer springs in the system, therefore the initial energy decreases when there are 50% or 75% selections. Also, as the objects become more independent, the time to stabilize the system decreases. For the extreme case in which no replica object is connected with another, the algorithm gets the stabile allocation in one iteration. The second time the replica objects are reintroduced into the queue, they are already at the stable location.

Scalability with Respect to the Update-to-Read Ratio

We ran experiments to see how the system reacts when the number of updates increases. We chose to have 10%, 20%, and 30% updates.

Even though we increased the number of updates to 30%, no merging occurred. Once again, this shows how highly replicable the data set is. However, the effect of increasing the number of updates is strongly felt, as in this test there are a one-third fewer new replicas than in the test where there are only 10% updates. Also, in the case of hot spots, the number of maximum replicas-per-fragment decreases to almost half.

Comparing these results with the ones from the previous experiment, we can see that the effects of changing the parameters had a similar influence on the data. The maximum number of replicas-per-fragment is almost identical with the one obtained in the previous experiment. So is the number of new created replicas. Increasing the number of updates from 10% to 20% gives similar results as decreasing the number of joins from 75% to 50%. It is the same situation when we consider the change from 20% to 30% updates and 50% to 75% selections. However, they create similar effects but in different ways. As mentioned, when increasing selectivity, fewer replica objects of different fragments are pulled towards

query objects (directly and indirectly), whereas when increasing the number of updates, the replica objects of the same fragment are forced to get closer to each other and eventually merge.

Scalability with Respect to the Query Access Distribution

In this test we compare the results of the 80/20 access distribution with those of the uniform distribution. There were more replicas created in the case of uniform distribution because the queries are evenly distributed to fragments, and the way we selected the value of σ_R. In the case of the 80/20 distribution, 80% of the queries access 20% of the data, which makes the hot fragments replicate at a high rate. However, as the threshold σ_R increases with the number of replicas, at some point not even the highly accessed fragments will replicate. Fragments that were cold in the 80/20 distribution are no longer cold in the uniform distribution, therefore they replicate. Since σ_R is gradually increasing with the number of replicas, its value is not hindering replication of a small number of replicas. Therefore, in the uniform distribution, more replicas are created overall (fewer replicas-per-fragment, but more fragments replicate). The maximum number of replicas each fragment has is smaller because the queries are now distributed and there are no hot fragments as in 80/20 distribution.

We were surprised to find that the maximum number of replicas that was generated for the 80/20 distribution is only slightly higher than for the uniform distribution. This was not expected, and we investigated the distribution of replicas to fragments. We have found that, in the case of uniform distribution, 14 replicas-per-fragment is an 'unlucky' exception. The reason is that, the fragment size was chosen randomly to be 1KB, but the size of the transferred data associated with the springs was much higher (1KB to 40KB for query and correlation springs). Therefore, the threshold σ_R was too small to stop the replication process. Out of the 200 fragments, one fragment had 14 copies, the second hottest had 10, four had 8 replicas, and five had 7 replicas. The large majority had 1, 2 or 3 copies.

Table 4.6: Performance Evaluation w.r.t. to S/J, U/R, and Query Access Distrib.

	Test	1	2	3	4	4	4	4	4
	Sel-to-Join(% Sel)	25	50	75	50	50	50	50	50
	Upd-to-Retr(% Upd)	20	20	20	10	20	30	20	20
	Access Distr.	80/20	80/20	80/20	80/20	80/20	80/20	80/20	Unif.
Init. Conf.	Nr.Replicas	377	377	377	377	377	377	377	377
	Rpl/Frg (min)	1	1	1	1	1	1	1	1
	Rpl/Frg (max)	4	4	4	4	4	4	4	4
	Rpl/Frg(mean)	1.89	1.89	1.89	1.89	1.89	1.89	1.89	1.89
	Rpl/Frg(med)	1	1	1	1	1	1	1	1
Final Conf.	Nr.Replicas	515	505	482	523	505	475	505	578
	Rpl/Frg (min)	1	1	1	1	1	1	1	1
	Rpl/Frg (max)	18	15	13	20	15	13	15	14
	Rpl/Frg(mean)	2.58	2.53	2.41	2.62	2.53	2.38	2.53	2.89
	Rpl/Frg(med)	2	2	2	2	2	2	2	3
Oper.	Moved	141	174	175	156	174	157	174	93
	Not Move	1916	2304	1760	1965	2304	1724	2304	2229
	Replicate	139	128	105	146	128	98	128	201
	Not Replicate	1517	1851	1395	1599	1851	1366	1851	1768
	Merged	1	0	0	0	0	0	0	0
Bubble Energy	Initial	5.06E7	4.51E7	3.81E8	4.55E8	4.51E7	4.23E7	4.51E7	4.33E7
	Final	3.80E7	3.31E7	2.85E7	3.27E8	3.31E7	3.25E7	3.31E7	3.21E7
	Running time	5:35	4:27	1:52	3:57	4:27	3:07	4:27	2:05

4.2.3 Conclusions

We have compared the Spring Algorithm with a limited exhaustive search algorithm to validate the proposed algorithm's efficacy. The cases in which the algorithm performed poorly were due to the fact that the tested environments were small. For these situations, placing even one object next to an optimal site can return poor results as each of the fragments in the system accounts for a significant part of the bubble energy. We showed that the reason for this is because it did not reconnect the springs when simulating a move. We argued that in a large scale system this would not be a problem. When running the Spring Algorithm, there is a tradeoff between the quality of results and run-time performance. We chose to have better run times because this algorithm should be used in large systems.

We have also shown that, when dealing with larger data sets, the algorithm scales well and the increase in the running time is usually logarithmic or, in the case of the number of queries, linear.

When we tested the system's reaction to increasing number of sites, we noticed that the number of replicas obtained in the end increased with the number of sites because more sites (20% of 1000) were highly accessed. This also causes the run time to increase. We compared the results when using 200 generated sites and 200 sites in a real setting and noticed that, because the notion of locality is better defined in the real setting, the fragments tended to move more and replicate less than in the case of generated settings.

In the case of fragments, we noticed that the ratio between the number of newly created replicas and the total number of fragments is indirectly proportional to the number of fragments. This is the result of sharing the same number of queries over an increasing number of fragments, which, on average, leads to less springs pulling from each object. Also, we noticed that in the systems with fewer fragments, the cost to compute the bubble energy is higher, because the energy of more fragments is recomputed when a fragment is relocated.

We showed that the number of created replicas increases logarithmically with the number of queries. The way we chose to compute σ_R for replication made replicating a fragment more and more difficult as its number of copies increased. Even though the size of the set of queries increased and the way we chose the system parameters favored replication, σ_R limited the number of performed replications.

Increasing the number of selections from 25% to 75% gives fragments more independence from each other. As a result, reallocating one fragment does not influence others, therefore the time to allocate a fragment decreases. Because there are more selections and fewer joins, the number of springs accessing each of the fragments also decreases. Less springs accessing each fragment results in fewer replicas created.

Increasing the number of updates from 10% to 30% caused an expected decrease in the number of newly created replicas. When comparing the effects of increasing the number of updates from 10% to 30% with those of increasing the selections from 25% to 75%, we noticed that all resulting parameters were similar. However, the processes performed to get the results were very different.

In the case of uniform distribution of queries to fragments, the number of created replicas increased more than in the 80/20 distribution. The fragments that were cold in the 80/20 distribution were accessed more, therefore they replicated. In 80/20 there were few fragments with a large number of replicas whereas the large majority of the fragments had only few replicas. In the uniform distribution, there was no discrepancy, as most fragments have the same number of copies.

Chapter 5

Conclusions and Future Work

5.1 Overview

Finding an efficient allocation is critical to ensuring cost-effective performance and high availability of data for Internet-scale database applications.

As discussed, most of the previous strategies are based on static, small distributed systems. As a result, they propose algorithms that are not able to react adaptively to changes in access patterns and do not consider the correlations between data fragments. None of the proposed strategies have analyzed the Internet-based data distribution problem in detail as we did, or their goals were different from our goal which is to minimize the overall access cost. Previous work does not consider the relations between the distributed data and the fact that changing the location of some data may affect other data. Also, some of the previous work tackles the distribution problem from a different perspective such as response time [14], bandwidth [47], or availability [36].

This thesis proposes a dynamic strategy to the allocation problem, suitable for a system running in the Internet environment that requires high adaptivity. Our algorithm is incremental and dynamic, executing data redistribution without knowledge of the global environment in advance, not running from scratch, and reacting dynamically to changing access patterns.

We modelled a distributed database system placed in the Internet environment that has three components: sites (with full DBMS functionalities), data (information requested by users), and network links (that connect sites). Data can be fragmented and replicated, and the unit of distribution is the fragment of a relation. Our system assumes an initial data distribution of fragments. The change in the way those fragments are accessed triggers their movement, replication, or deletion (the system reconfigures itself dynamically). The

sites cooperate to improve the overall cost of accessing data, and decide when and where to move or replicate a replica object.

Each pair of sites has a communication cost associated to it. The access cost between two sites represents the distance between them (the delay when sending 1KB of data between two sites). To get good system performance, data should be close to the user to allow a low communication cost. Also, to reduce the consistency costs, data should not be kept on sites where it is not accessed by users. As our goal is to minimize the overall access cost to fragments, all these individual costs become constraints that we have to deal with when addressing the allocation problem.

We named our strategy the **spring system approach** as the dynamic system we modelled is based on the same principles as a physical spring system – it had similar components and interactions:

- The objects in the spring system are, in our case, relation fragments (replica objects) that need to be redistributed, and queries (query objects) that access the replica objects. The replica objects are mobile, whereas the query objects are fixed.

- Relationships between objects are simulated by springs connecting the two types of objects. We use different types of springs: (a) a query spring that connects a query object to a replica object; (b) a correlation spring that connects two replica objects of different fragments (due to join operations); (c) a replication spring that connects two replica objects of a fragment. These three types of springs simulate all situations that we model.

  In our system, the closer the replica objects are to the query object, the lower the cost of executing the query, which is defined as the sum of the energies of the springs connecting the objects it accesses.

- The spring characteristic that interests us most is stretchability. Whenever the spring is taken out of its equilibrium state, there is a force that wants to pull it back to its original position.

  In the spring system, an object that is directly connected to an object that moves is more affected than objects that are not directly connected. Objects stop moving when they reach equilibrium, i.e. when the objects have the minimum energy. This corresponds to the case in which no object has kinetic energy (they no longer move), and the potential energy (the energy in the springs) is minimum. The equilibrium state represents the allocation where the replica objects should be distributed in order to get the best system performance.

- The bubble represents the part of the spring system that is not affected by the changes that occur in another part. The spring system is composed of bubbles. There are two edge cases here: first, the whole spring system is one big bubble. In the Internet environment this is never the case. On the other extreme, each replica object can represent a bubble and the spring system is made of many small bubbles.

The Spring Algorithm is a dynamic incremental algorithm that performs the replication and redistribution of fragments based on the initial configuration given by the system we modelled. The algorithm is triggered by the change in the access pattern of an object (change in the fragment's update frequency; or data being more/less popular). The Spring Algorithm checks whether the best performance is obtained by moving or replicating the object. Since only the neighbor sites are analyzed, an object that has been moved is reanalyzed, as this can lead to a better location. If an object has been replicated, both objects are reanalyzed. The new potential candidate sites are now the neighbor sites of the new location. The object stops moving when no better local site is found. Each object has been analyzed based on the fixed allocation of the other objects in the bubble. Since an object could have been considered well allocated and eliminated from the queue based on a bad configuration, one iteration of the algorithm may not give a good allocation configuration. When an iteration of the algorithm stops, all objects are reintroduced into the queue and the algorithm is re-run. The reallocation stops when the bubble energy obtained in the nth iteration has decreased less than a specified threshold in comparison with previous iteration.

To analyze the performance of the Spring Algorithm, we implemented a simulation that performed two sets of experiments:

- The validation experiments compared the Spring Algorithm with a limited exhaustive search algorithm.
- The performance evaluation experiments verified the system's scalability by varying several parameters.

When comparing the Spring Algorithm with the limited exhaustive search algorithm, we noticed that our algorithm returned good results except for three cases. Those cases were due to the small testing environment where each fragment had about 20% of the allocated bubble energy. In these three cases, the cost was severely influenced by: (a) the order in which the replica objects were analyzed; (b) the algorithm not reconnecting the springs when simulating a move, but only when simulating a replication; and (c) the number of neighbor sites that were analyzed. In the Internet environment those should not be issues that will decrease performance.

The performance evaluation results are encouraging. We tested the algorithm to see how it reacts when various parameters are modified. We used various number of sites, fragments, and queries, various update-to-retrieval and selection-to-join ratios, and different distributions of queries to data. We found that the algorithm scales well when the number of replicas and the number of sites is increased, with the running time increasing logarithmically. The increase in the number of queries results in a linear increase in the running time. However, we believe that this is not an issue because the algorithm can be distributed as presented in the next section. Regarding the update-to-retrieval ratio, the system responded as expected by reducing the number of replicas created with an increase of the number of updates (due to the increase in cost). The selection-to-join ratio affected the fragments' independence. A lower percentage of selections resulted in more correlation springs that generated more replicas. The uniform distribution of queries to fragments resulted in a larger number of total replicas, but the maximum number of replicas any fragment had decreased when compared to the 80/20 distribution.

5.2 Contributions

This thesis takes into consideration that, in a DDBMS, groups of fragments have a tendency to be accessed together by introducing the concept of a "bubble". This is the minimum unit of data that should be analyzed independently because the fragments it contains are strictly related and they should be analyzed together. Since operations in a bubble do not affect other bubbles, the bubble concept also helps when performing the allocation. The analysis is performed only on related fragments, which reduces the time to perform the allocation. The independence of a bubble also helps us minimize the overall energy of the spring system by minimizing the energy of each bubble in the system.

Unlike previous works, we consider the correlations between fragments. Considering fragment interdependencies made the problem more challenging as simple greedy-like algorithms could not be applied, requiring the development of a new dynamic algorithm.

Because we address the Internet environment which is dynamic and incremental, we specially tailored the algorithm to apply to this environment. This means that we expect the changes to be incremental, therefore they would not affect the allocation configuration substantially. We consider the changes to be not substantial when the objects change their allocation but remain in the neighborhood. For this reason and in this particular situation, the running time is much lower than everything proposed before: $O(n \cdot v)$, where n is the number of fragments to be allocated and v is the number of neighbor sites (fixed to 10) that are analyzed for move/replication.

The most substantial contribution is introducing the "spring system" analogy to per-

form data redistribution. This approach helped us visualize the distribution problem and gave us ideas about how to create an algorithm for a dynamic environment where changes are incremental. We have shown that the spring system interactions match the fragment interactions in real DDBMSs and that following the way the spring system stabilizes leads to a very good data allocation of fragments. The analogy is very natural and we have shown that there are only three points when the Spring Algorithm did not perfectly match the distribution problem:

- One situation was when fragment replication was performed. For that case, in the spring system, an object and additional replication springs had to be created. Also, the springs that were connected to the initial copy are now divided between the two copies.
- The second situation was when two replicas of a fragment were merged. For that case, in the spring system, one object had to be removed and the springs that were connected to the removed object are now connected to the remaining one.
- The third situation is represented by the case in which the replica objects "jump" between neighbor sites. In a real spring system, the objects can move everywhere and there is no notion of site.

After any of the two operations that are unnatural for a spring system are performed, the new and the old copies find their way to the best site where they should be placed without further intervention.

5.3 Future Work

As future work, we identified four issues that should be further investigated.

Multi-fragment analysis

The algorithm we propose finds a better allocation of fragments by analyzing one replica object at a time, while the rest are fixed. However, if two or more replica objects are strongly connected, it may be difficult to move them one at once. As a short term future work we would like to change to algorithm to detect when a set of replica objects are strongly connected and try to move them together.

Parallelism

The weakness of the current implementation of the Spring Algorithm is that it is centralized. As a result, we have seen that, when increasing the number of queries, the running

time increases linearly. As future work, we would like to focus on implementing a P2P system on which to test the algorithm. In this environment, the workload would be distributed to multiple sites, reducing the response time. The time needed to stabilize the system would represent the time elapsed until each site had finished running the algorithm. If the time to stabilize the system when there are q queries is t, the time to stabilize it when there are double the number of queries is $t + q/n$, where n is the number of sites in the system and uniform distribution is assumed.

Each site would decide when to move or replicate an object it stores. Based on the accesses to the objects it stores, the site computes the access cost if the object is to be placed on one of the neighbor sites. This could be performed by sending representative statistics of the object's access pattern to all neighbor sites. Each site would return the cost of accessing the replica object if the object is located at that site. The object would be moved to the site that returned the lowest cost, if that cost is lower than the current one. After that, the selected site would deal with the replica object. If the object is already at the best location found by the algorithm and its access frequency has increased, the object would be tested for replication. In this case, the query objects accessing the initial object would be divided between the two objects, whichever is closer, creating new access patterns for both of them.

Another possibility is to take advantage of the fact that each bubble is independent and have a site responsible for reallocation. The site can be chosen using an election algorithm.

The results obtained from a simulation performed on a P2P system would better reflect the reality. Also, due to the parallelism, the number of sites, fragments and queries can be increased with few orders of magnitude, truly reflecting the scale of the Internet-based applications.

Increasing Performance by Re-fragmentation

In a highly dynamic environment such as the Internet, not only do the access patterns to fragments changes over time, but also the nature of access to fragments. Because we deal with database fragments, both fragmentation and data allocation should be performed together in order to get optimal results. In the spring system approach we assume that fragmentation is already performed. Future work would focus on creating a new module of the Spring Algorithm that would perform data re-fragmentation.

A Different Heuristic

The Spring Algorithm works by placing the moved or replicated replica objects at the end of the the analysis queue. The replica objects that were not moved or replicated are removed from the queue and never reanalyzed. As we mentioned, replica objects are connected and moving one affects others. We counterbalance this issue by running multiple

iterations of the algorithm. However, with each iteration, the Spring Algorithm analyzes the replica objects that are already at the best location, which results in higher running times.

A short term future work would focus on developing an algorithm in which, when a replica object is moved, the replica objects directly connected to it are also placed at the end of the queue (if they are not in the queue already). For this case it would be enough to run the algorithm once. On one hand, we think the new algorithm would perform better only if there are many independent fragments. These are inserted at the end of the queue, redistributed and then removed. Because they would be analyzed once, which will lead to a shorter running time. On the other hand, we assume that, if the replicas are connected, the two algorithms would have a similar performance. What the Spring Algorithm does by running multiple times would be similar to what the second algorithm would do by reintroducing the connected replicas into the queue.

Bibliography

[1] Akamai Technologies, http://www.akamai.com.

[2] Mirror Image Internet, http://mirror-image.com.

[3] SinoCDN, http://www.sinocdn.com.

[4] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract). In *Euro-Par '97 Parallel Processing, 3rd Int. Euro-Par Conf.*, pages 496–503, 1997.

[5] F. Baiao, M. Mattoso, and G. Zaverucha. Horizontal Fragmentation in Object DBMS: New Issues and Performance Evaluation. In *Proc. of the 19th IEEE Int. Performance, Computing and Communications Conf.*, pages 108–117, 2000.

[6] D. Barkai. Technologies for Sharing and Collaborating on the Net. In *1st Int. Conf. on Peer-to-Peer Computing*, pages 13–28, 2001.

[7] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data Management for Peer-to-Peer Computing: a Vision. In *Proc. 5th Int. Workshop on the World Wide Web and Databases (WebDB)*, pages 89–94, 2002.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1st edition, 1987.

[9] A. Bestavros and C. Cunha. Server-Initiated Document Dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19:3–11, 1996.

[10] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update Propagation Protocols for Replicated Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 97–108, 1999.

[11] A. Brunstrom, S. T. Leutenegger, and R. Simha. Experimental Evaluation of Dynamic Data Allocation Strategies in A Distributed Database with Changing Workloads. In *Proc. 4th Int. Conf. on Information and Knowledge Management*, pages 395–402, 1995.

[12] C.W. Cameron, S.H. Low, and D.X. Wei. High-Density Model for Server Allocation and Placement. In *ACM SIGMETRICS 2002*, pages 152–159, 2002.

[13] T.S. Chen, C.Y. Chang, J.P. Sheu, and G.J. Yu. A Fault-Tolerant Model for Replication in Distributed File Systems. In *Proc. of National Science Counc. R.O.C.*, volume 23, pages 402–410, 1999.

[14] Y. Chen, R.H. Katz, and J.D. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *1st Int. Workshop on Peer-to-Peer Systems*, pages 306–318, 2002.

[15] S.A. Cook, J.P., and I.S. Pressman. The Optimal Location of Replicas in a Network Using a READ-ONE-WRITE-ALL Policy. *Distributed Computing*, 15(1):57–66, 2002.

[16] B. Cooper and H. Garcia-Molina. Peer-to-Peer Resource Trading in a Reliable Distributed System. In *1st Int. Workshop on Peer-to-Peer Systems*, pages 319–327, 2002.

[17] A.L. Corcoran and J. Hale. A Genetic Algorithm for Fragment Allocation in a Distributed Database System. In *Proc. 1994 Symp. on Applied Computing*, pages 247–250, 1994.

[18] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 3rd edition, 2000.

[19] K. Eswaran. Placement of Records of a File and File Allocation in Computer Networks. In *Proc. of 1974 Int. Federation for Information Processing*, pages 304–307, 1974.

[20] P. Felber and A. Schiper. Optimistic Active Replication. In *Proc. 21st Int. Conf. on Distributed Computing Systems*, pages 333–341, 2001.

[21] D.E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison Wesley, 1st edition, 1989.

[22] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, 1996.

[23] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What Can Databases Do for Peer-to-Peer? In *Proc. 4th Int. Workshop on the World Wide Web and Databases (WebDB)*, pages 31–36, 2001.

[24] J.S. Gwertzman and M. Seltzer. The Case For Geographical Push-Caching. In *Proc. of the Workshop on Hot Topics in Operating Systems*, pages 51–57, 1995.

[25] J. Hall, J. D. Hartline, A. R. Karlin, J. Saia, and J. Wilkes. On Algorithms for Efficient Data Migration. In *Proc. of the 12th ACM-SIAM Symp. on Discrete Algorithms*, pages 620–629, 2001.

[26] Y.F. Huang and J.H. Chen. Fragment Allocation in Distributed Database Design. *Journal of Information Science and Engineering*, 17(3):491–506, 2001.

[27] J. Sidell and P.M. Aoki and A. Sah and C. Staelin and M. Stonebraker and A. Yu. Data Replication in Mariposa. In *Proc. 12th Int. Conf. on Data Engineering*, pages 485–494, 1996.

[28] J.M. Johansson, S.T. March, and J.D. Naumann. The Effects of Parallel Processing on Update Response Time in Distributed Database Design. In *Proc. of the 21st Int. Conf. on Information systems*, pages 187–196, 2000.

[29] J. Kangasharju, J. Roberts, and K. Ross. Object Replication Strategies in Content Distribution Networks. In *Computer Communications*, volume 25, pages 367–383, 2002.

[30] K. Kant, R. Iyer, and V. Tewari. A Framework for Classifying Peer-to-Peer Technologies. In *2nd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, pages 368–375, 2002.

[31] M. Karlsson and M. Mahalingam. Do We Need Replica Placement Algorithms in Content Delivery Networks? In *7th Int. Workshop on Web Content Caching and Distribution*, pages 117–128, 2002.

[32] B. Kemme and A. Bartoli. Recovering from Total Failures in Replicated Databases. Technical report, Trieste University, 2003.

[33] M. Khair, I. Mavridis, and G. Pangalos. Design of Secure Distributed Medical Database Systems. In *Database and Expert Systems Applications, 9th Int. Conf.*, pages 492–500, 1998.

[34] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman. Placement Algorithms for Hierarchical Cooperative Caching. In *Proc. of the 10th ACM-SIAM Symp. on Discrete Algorithms*, pages 586–595, 1999.

[35] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[36] B. Li, M. Golin, G. Italiano, and X. Deng. On the Optimal Placement of Web Proxies in the Internet. In *Proc. of IEEE Infocom*, pages 1282–1290, 1999.

[37] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proc. 21st Int. Conf. on Distributed Computing Systems*, pages 233–242, 2002.

[38] W.S. Ng, B.C. Ooi, K.L. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[39] M. Noronha. An Introduction to Data Fragmentation in Informix Dynamic Server. *IBM Library*, 2002.

[40] C. Olston and J. Widom. Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 144–155, 2000.

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

[42] L. Qiu, V.N. Padmanabhan, and G.M. Voelker. On the Placement of Web Server Replicas. In *Proc. of IEEE Infocom*, pages 1587–1596, 2001.

[43] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. In *Proc. 19th Int. Conf. on Distributed Computing Systems*, pages 101–113, 1999.

[44] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving Data Availability through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities. In *Global and Peer-to-Peer Computing on Large Scale Distributed Systems Workshop*, pages 376–381, 2002.

[45] A.S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 1st edition, 2002.

[46] A. Thomasian. *Performance Analysis of Database Systems*, pages 305–327. Lecture Notes in Computer Science. Springer Verlag, 2000.

[47] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth Constrained Placement in a WAN. In *Proc. ACM SIGACT-SIGOPS 20th Symp. on the Principles of Dist. Comp.*, pages 134–143, 2001.

[48] S. Voulgaris, M. van Steen, A. Baggio, and G. Ballintijn. Transparent Data Relocation in Highly Available Distributed Systems. In *6th Int. Conf. On Principles Of Distributed Systems*, 2002.

[49] O. Wolfson and S. Jajodia. An Algorithm for Dynamic Data Distribution. In *IEEE Workshop on Management of Replicated Data*, pages 62–65, 1992.

[50] O. Wolfson, S. Jajodia, and Y. Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[51] H. Yu and A. Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In *The VLDB Journal*, pages 123–133, 2000.