Distrib Parallel Databases (2010) 28: 157–185
DOI 10.1007/s10619-010-7068-1

DYFRAM: dynamic fragmentation and replica management in distributed database systems

Jon Olav Hauglid · Norvald H. Ryeng · Kjetil Nørvåg

Published online: 8 September 2010

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

Keywords Distributed DBMS · Fragmentation · Replication · Physical database design

Communicated by Kam-Fai Wong.

Supported by grant #176894/V30 from the Norwegian Research Council.

J.O. Hauglid · N.H. Ryeng · K. Nørvåg (✉)
Dept. of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway
e-mail: noervaag@idi.ntnu.no

J.O. Hauglid
e-mail: joh@idi.ntnu.no

N.H. Ryeng
e-mail: ryeng@idi.ntnu.no

1 Introduction

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

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

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

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

An example of what we aim at achieving with our approach is illustrated in Fig. 1. It illustrates the access pattern of a database table from two sites. Site 1 has a uniform distribution of accesses, while site 2 has an access pattern with distinct hot spots. In this case, a good fragmentation would generate 6 fragments, one for each of the hot spot areas and one for each of the intermediate areas. A good allocation would be the fragments of the hot spot areas (F_1, F_3, and F_5) allocated to site 2, with the other fragments (F_2, F_4, and F_6) allocated to site 1. As will be shown later in the experimental evaluation, DYFRAM will detect this pattern, split the table into appropriate

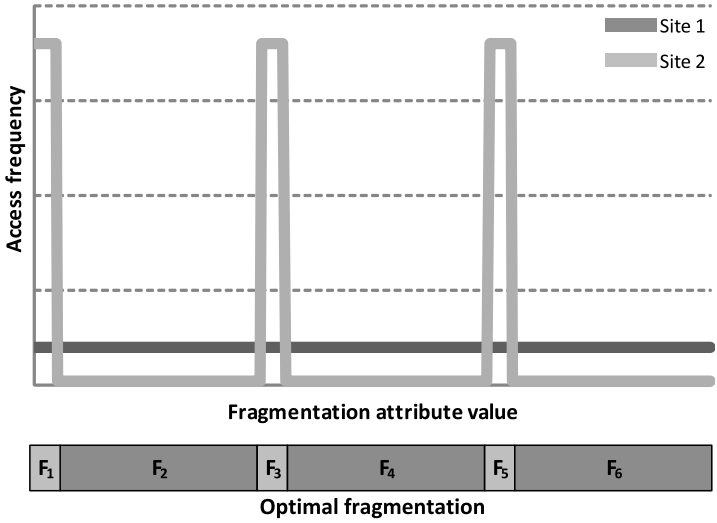


Fig. 1 Example access pattern, and desired fragmentation and allocation

fragments, and then allocate these fragments to the appropriate sites. Whether some of the fragments should be replicated or not depends on the read/write pattern. Note that if the access pattern changes later, this will be detected and fragments reallocated as well as repartitioned if necessary.

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

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

2 Related work

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

Fragmentation is tightly coupled with fragment allocation. There are methods that do only fragmentation [1, 24, 26, 33, 34] and methods that do only allocation of predefined fragments [3, 4, 7, 10, 13, 20, 30]. Some methods also exist that integrate both tasks [9, 11, 17, 19, 25, 27, 29]. Replication, however, is typically done as a separate task [5, 8, 15, 21, 22, 32], although some methods, like ours, take an integral view of fragmentation, allocation and replication [11, 27, 29]. Dynamic replication algorithms [5, 15, 21, 22, 32] can optimize for different measures, but we believe that refragmentation and reallocation must be considered as alternatives to replication. In DYFRAM we choose among all these options when optimizing for communication costs. Our replication scheme is somewhat similar to that of DIBAS [11], but DYFRAM also allows remote reads and writes to the master replica, whereas DIBAS always uses replication for reads and do not allow remote writes to the master replica. This operation shipping is important when analyses [8] of replication vs. remote reads and writes conclude that the replication costs in some cases may be higher than the gain from local data access. A key difference between DIBAS and DYFRAM is that DIBAS is a static method where replication is based on offline analysis of database accesses, while DYFRAM is dynamic and does replication online as the workload changes.

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

Static methods look at a set of queries or operations. It can be argued that the workload should be viewed as a sequence of operations, not as a set [2]. Dynamic methods continuously monitor the database and adapt to the workload as it is at the moment and are thus viewing a sequence of operations. Dynamic methods are part of the trend towards fully automatic tuning [31], which has become a popular research direction. Recently, work has appeared aiming at integrating vertical and physical partitioning while also taking other physical design features like indices and materialized views into consideration [1]. Adaptive indexing [2, 6] aims to create indices dynamically when the costs can be amortized over a long sequence of read operations, and to drop them if there is a long sequence of write operations that would suffer from having to

update both base tables and indices. Our work is on tables and table fragments, but shares the idea of amortizing costs over the expected sequence of operations. In adaptive data placement, the focus has either been on load balancing by data balancing [9, 17], or on query analysis [19]. In our algorithms, we seek to place data on the sites where they are being used (by reads or writes), not to balance the load.

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

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

Among the decentralized systems, we find replication schemes for mobile ad hoc networks (see [23] for an overview). However, these approaches do not consider table fragmentation and in general do replication decisions on a more coarse granularity, e.g., files.

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

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

A summary and feature comparison of our method and related fragmentation, allocation and replication methods is given in Table 1. We show which features are provided by each method and whether it is a dynamic method that adapts to the workload or a static method that never updates its decision. The methods are also

Table 1 Summary of related fragmentation, allocation and replication methods

	Fragmentation	Allocation	Replication	Dynamic	Static	Centralized	Decentralized
DYFRAM	✓	✓	✓	✓			✓
Agrawal et al. [1]	✓				✓	✓	
Ahmad et al. [3]		✓			✓	✓	
Apers [4]		✓			✓	✓	
Bonvin et al. [5]			✓	✓			✓
Brunstrom et al. [7]		✓		✓			✓
Ciciani et al. [8]			✓		✓	✓	
Copeland et al. [9]	✓	✓		✓		✓	
Corcoran and Hale [10]		✓			✓	✓	
Didriksen and Galindo-Legaria [11]	✓	✓	✓		✓	✓	
Furtado [13]		✓			✓	✓	
Hara and Madria [15]			✓	✓			✓
Hua and Lee [17]	✓	✓		✓		✓	
Ivanova et al. [19]	✓	✓		✓		✓	
Menon [20]		✓			✓	✓	
Mondal et al. [21]			✓	✓			✓
Mondal et al. [22]			✓	✓			✓
Rao et al. [24]	✓				✓	✓	
Saccà and Wiederhold [25]	✓	✓			✓	✓	
Shin and Irani [26]	✓				✓	✓	
Sidell et al. [27]	✓	✓	✓	✓			✓
Tamhankar and Ram [29]	✓	✓	✓		✓	✓	
Ulus and Uysal [30]		✓		✓			✓
Wolfson and Jajodia [32]			✓	✓			✓
Wong and Katz [33]	✓				✓	✓	
Zilio et al. [34]	✓				✓	✓	

categorized according to the where the decisions to fragment, allocate and replicate are made. This can be done either centralized to a single site which has the necessary information about the other sites, or decentralized.

3 Preliminaries

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

Table 2 Symbols

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

3.1 System model

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

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

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

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

3.2 Fragment model

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

adjacent if their *FVD* meets, i.e.:

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

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

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

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

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

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

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

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

3.3 Problem definition

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

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

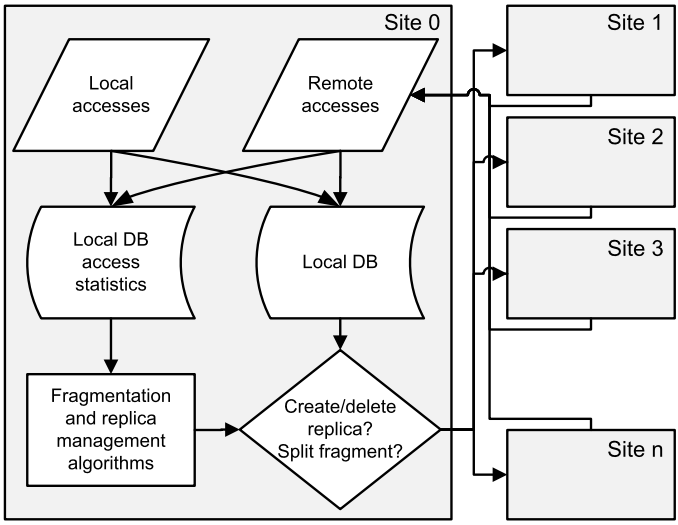


Fig. 2 Dynamic fragmentation and allocation

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

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

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

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

4 Overview of DYFRAM

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

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

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

Given the available statistics, our algorithm examines accesses for each replica and evaluates possible refragmentations and reallocations based on recent history. The algorithm runs at given intervals, individually for each replica. Since decisions are made independently of other sites, decisions are made based on the information available at that site. With master-copy based replication, all writes are made to the master replica before read replicas are updated. Therefore, write statistics are available at all sites with a replica of a given fragment. On the other hand, reads are only logged at the site where the accessed replica is located. This means that read statistics are spread throughout the system. In order to detect if a specific site has a read pattern that indicates that it should be given a replica, we require a site to read from a specific replica so that this site's read pattern is not distributed among several replicas.

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

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

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

5 Replica access statistics

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

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

– Must hold enough information to capture read and write patterns.

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

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

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

5.1 Histogram design

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

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

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

Histogram-related symbols used in the following discussion are summarized in Table 3. Each bucket in a histogram H_i has a bucket number b_k and contains two values: the read count $R_i[b_k]$ and the write count $W_i[b_k]$. We use equi-width histograms with bucket width W and limit bucket value ranges to start and end on multiples of W. The value range of a bucket is then $[b_k \cdot W, (b_k + 1) \cdot W)$.

Histograms only maintain statistics for values that are actually accessed, i.e., they do not cover the whole *FVD*. This saves space by not storing empty buckets, which

Table 3 Histogram symbols

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

is useful since we lack a priori knowledge about fragment attribute values. Buckets are therefore stored as $(b_k, R_i[b_k], W_i[b_k])$ triplets hashed on b_k for fast access.

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

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

5.2 Histogram operations

This section presents algorithms for the different histogram operations.

5.2.1 Histogram update

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

Since this operation is performed very often, it is important that it is efficient. As described above, the value range of bucket number b_k is $[b_k \cdot W, (b_k + 1) \cdot W)$. We therefore need to determine b_k for a given fragmentation attribute value v_j and then

Algorithm 1 Site S_i reads tuple t_j in replica R_j with fragmentation attribute value v_j. (Similar for writes.)

```
histogramUpdate(S_i, R_j, v_j):
  H_i ← H_cur[S_i, R_j]
  b_k ← v_j / W
  R_i[b_k] ← R_i[b_k] + 1
  if numberOfBuckets > MAX_B then
    increaseBucketWidth(R_j)
  end if
```

increment its bucket value. The formula is $b_k = v_j/W$, which means that the computational cost is $O(1)$. Also, since histograms are kept in main memory, histogram updates do not incur any disk accesses.

If no bucket already exists for bucket number b_k, a new bucket must be constructed. This is the only time where the histogram gets more buckets, so after the update, the current number of buckets is checked against the upper bound MAX_B and bucket width is increased (and thus the number of buckets decreased) if we now have too many buckets.

5.2.2 *Histogram bucket resizing*

If at any time a tuple access occurs outside the range covered by the current buckets, a new bucket is made. If the upper bound of buckets, MAX_B, is reached, the bucket width W is increased and the histograms reorganized. We do this by multiplying W with a scaling factor Z_W. This factor is an integer such that the contents of new buckets are the sum of a number of old buckets. Increasing bucket width of course reduces the histogram accuracy, but it helps reduce both memory usage and processing overhead. Since we only store recent history, we may reach a point where the set of buckets in use becomes very small. If we can reduce bucket width to W/Z_W and still have fewer buckets than the upper bound, the histogram is reorganized by splitting each bucket into Z_W new buckets. This reorganization assumes uniform distribution of values inside each bucket, which is a common assumption [18]. Details are shown in Algorithm 2. Note that this is performed for both the current and old set of histograms in order to make them have the same bucket width, as this makes subsequent histogram accesses efficient. The function *getActiveSites*(R) returns the set of all sites that have accessed replica R.

Similarly, if we at any point use only a very low number of buckets, the bucket widths can be decreased in order to make access statistics more accurate. This is described in Algorithm 3. Of special note is the expression $max(1, R_i[b_k]/Z_W)$. If a large bucket to be divided into smaller buckets contain only a few tuples, rounding can make $R_i[b_k]/Z_W = 0$, which would in effect remove the bucket (since only buckets containing tuples are stored). To prevent loss of information in this case, new buckets contain a minimum of 1 tuple.

Algorithm 2 Increase bucket width W for histograms for replica R by factor Z_W.

```
increaseBucketWidth(R):
  for all S_i ∈ getActiveSites(R) do
    for all H_i ∈ H_cur[S_i, R] ∪ H_old[S_i, R] do
      H'_i ← ∅
      for all b_k ∈ H_i do
        b'_k = b_k / Z_W
        R'_i[b'_k] = R'_i[b'_k] + R_i[b_k]
        W'_i[b'_k] = W'_i[b'_k] + W_i[b_k]
      end for
      H_i ← H'_i
    end for
  end for
```

5.2.3 Histogram range count

When retrieving access statistics from histograms, i.e., contents of buckets within a range, both current and old histograms are used. Since both histograms have the same bucket width and corresponding bucket numbers, retrieval is a straight summation of range counts from the two histograms and therefore very fast to perform. In order to count number of reads or writes from site S to replica R stored in buckets numbered $[b_{min}, b_{max}]$, the functions *histogramReadCount*(S, R, b_{min}, b_{max}) and *histogramWriteCount*(S, R, b_{min}, b_{max}) are used. In order to get the sum of range counts for writes from all sites, the function *histogramWriteCountAll*(R, b_{min}, b_{max}) is used.

5.2.4 Histogram reorganization

As stated earlier, it is important that only the recent access history is used for replica evaluations in order to make it possible to adapt to changes in access patterns. This is achieved by having two sets of histograms, one current histogram H_{cur} that is maintained and one H_{old} which contains statistics from the previous period. Periodically the current H_{old} is replaced with the current contents of H_{cur}, and then H_{cur} is emptied and subsequently used for new statistics.

The only time buckets are removed from the histogram is during reorganization. It is therefore the only time that the number of buckets in the histogram can get so low that we can decrease the bucket width (thus creating more buckets) and still stay below the bucket number maximum MAX_B. This will be performed using *decreaseBucketWidth*(R) described in Algorithm 3. The function performing the reorganization is in the following denoted *histogramReorganize*(R).

5.3 Histogram memory requirements

wrong, remainders from old version It is important that the size of the histograms is small so that enough main memory is available for more efficient query processing and buffering. For every replica a site has, it must store two histograms for each

Algorithm 3 Decrease bucket width W for histograms for replica R by factor Z_W.

```
decreaseBucketWidth(R):
  for all S_i ∈ getActiveSites(R) do
    for all H_i ∈ H_cur[S_i, R] ∪ H_old[S_i, R] do
      H'_i ← ∅
      for all b_k ∈ H_i do
        for b'_k = 0 to Z_W do
          R'_i[b_k · Z_W + b'_k] = max(1, R_i[b_k]/Z_W)
          W'_i[b_k · Z_W + b'_k] = max(1, W_i[b_k]/Z_W)
        end for
      end for
      H_i ← H'_i
    end for
  end for
```

active site accessing the fragment. Every bucket is stored as a $(b_k, R_i[b_k], W_i[b_k])$ triplet (note that sparse histograms are used, so that only buckets actually accessed are stored). Assuming b buckets and c active sites, the memory requirement for each replica is $2 \cdot c \cdot b \cdot sizeOf(bucket)$ or $O(b \cdot c)$. Since b have an upper bound MAX_B, memory consumption does not depend on fragment size or number of accesses, only on the number of active sites.

6 Fragmentation and replication

Our approach calls for three different algorithms. One for creating new replicas, one for deleting replicas and one for splitting and coalescing fragments. These will be described in the following sections.

These algorithms are designed to work together to dynamically manage fragmentation and replication of those fragments such that the overall communication costs are minimized. The communication cost consists of four parts: (1) remote writes, (2) remote reads, (3) updates of read replicas, and (4) migration of replicas (either in itself or as part of creation of a new replica).

Common for all three algorithms is that they seek to estimate the benefit from a given action based on available usage statistics. This is implemented using three cost functions, one for each algorithm. These functions are described in Sect. 6.4.

6.1 Creating replicas

This algorithm is run at regular intervals for each fragment of which a given site has a replica. The aim is to identify sites that, based on recent usage statistics, should be assigned a replica. If any such sites are found, replicas are sent to them, and the site holding the master replica is notified so that the new replicas can receive updates.

The algorithm for identifying and creating new replicas of replica R is shown in Algorithm 4. In the algorithm, a cost function (to be described in Sect. 6.4) is applied

Algorithm 4 Evaluate replica R for any possible new replicas. R is located on site S_l.

```
createReplica(R):
  b_min ← min(H_cur[S_l, R]) {First bucket used}
  b_max ← max(H_cur[S_l, R]) {Last bucket used}
  card_w ← histogramWriteCountAll(R, b_min, b_max)
  for all S_r ∈ getActiveSites(R) do
    card_r r ← histogramReadCount(S_r, R, b_min, b_max)
    utility ← w_BE · card_r r − card_w − w_FS · card(R)
    if utility > 0 then
      copyReplica(R, S_r) {Also notifies master replica}
    end if
  end for
```

Algorithm 5 Evaluate local replica R and decide if it should be deleted. R is located on site S_l.

```
deleteReplica(R):
  b_min ← min(H_cur[S_l, R]) {First bucket used}
  b_max ← max(H_cur[S_l, R]) {Last bucket used}
  card_w ← histogramWriteCountAll(R, b_min, b_max)
  card_l r ← histogramReadCount(S_l, R, b_min, b_max)
  utility ← w_BE · card_w − card_l r
  if utility > 0 then
    deleteLocalReplica(R) {Also notifies master replica}
  end if
```

for each remote site S_r that has read from to R. The result is a *utility value* that estimates the communication cost reduction achieved by creating a new replica at site S_r. All sites with positive utility value receive a replica. If no site has a positive utility, no change is made.

Note that, if desired, the number of replicas in the system can be constrained by having a limit on the number of replicas. This might be beneficial in the context of massive read access to various sites.

6.2 Deleting replicas

Since each fragment must have a master replica, only read replicas are considered for deletion. This algorithm evaluates all read replicas a given site has, in order to detect if the overall communication cost of the system would be lower if the replica were deleted. The details are shown in Algorithm 5. Again, a cost function is used to evaluate each read replica R. Any replica with a positive utility is deleted after the site with the master replica has been notified.

6.3 Splitting fragments

The aim of the fragmentation algorithm is to identify parts of a table fragment that, based on recent history, should be extracted to form a new fragment and migrated to a different site in order to reduce communication costs (denoted *extract+migrate*). To avoid different fragmentation decisions made simultaneously at sites with replicas of the same fragment, this algorithm is only applied to master replicas.

More formally, assume a fragmentation $\mathcal{F}_{old}$ which includes a fragment F_i with $FVD(F_i) = F_i[min_i, max_i]$ having master replica R_i^m allocated to site S_i. Find a set of fragments $F_m, \ldots, F_n$ such that $\bigcup F_m, \ldots, F_n = F_i$ with $F_{new} \in F_m, \ldots, F_n$ and master replica R_{new}^m allocated to site $S_k \neq S_i$ such that the communication cost $C_{total} = \sum C(A_i) + \sum C(RE_j)$ is lower than for $\mathcal{F}_{old}$.

The result of each execution can be either: (1) do nothing, i.e., the fragment is as it should be, (2) migrate the whole master replica, or (3) extract a new fragment F_{new} with $FVD(F_{new}) = F_{new}[min_{new}, max_{new}]$ and migrate its new master replica to site S_k. A decision to migrate the whole master replica can be seen as a special case of extract+migrate. In the discussion below, we therefore focus on how to find appropriate values for min_{new} and max_{new}. If a refragmentation decision is made, all sites with read replicas are notified so that they can perform the same refragmentation. This is necessary to enforce that all replicas of a given fragment are equal.

The algorithm for evaluating and refragmenting a given fragment F is presented in Algorithm 6. It evaluates all new possible fragments F_{new} and possible recipient sites S_r using a cost function. The result is a utility value that estimates the communication cost reduction from extracting F_{new} and migrating its master replica to S_r. Afterward, all *compatible* fragmentations with positive utility values are performed. Two fragmentations are compatible if their extracted fragments do not overlap. In case of two incompatible fragmentations, the fragmentation with the highest utility value is chosen. Note that no fragments with *FVD* less than *fragmentMinSize* will be extracted in order to prevent refragmentation from resulting in an excessive number of fragments.

Given a fragment F_i with $FVD(F_i) = F_i[min_i, max_i]$, the size of the fragment value domain is then $width = max_i - min_i + 1$. Assume an extraction of a new fragment F_{new} such that $FVD(F_{new}) = F_{new}[min_{new}, max_{new}] \in FVD(F_i)$. If $FVD(F_{new})$ is assumed to be non-empty, i.e., $max_{new} > min_{new}$, then $width - 1$ possible values for min_{new} and max_{new} are possible. This means that $O(width^2)$ possible fragments F_{new} will have to be evaluated. This could easily lead to a prohibitively large number of F_{new} to consider, so some heuristic is required.

We reduce the number of possible fragments to consider based on the following observation: The basis for the evaluation algorithm is the access histograms described above. These histograms represent an approximation since details are limited to the histogram buckets. It is therefore only meaningful to consider $FVD(F_{new})$ with start/end-points at histogram bucket boundaries.

With b histogram buckets and $b \ll width$ as well as b having an upper bound, processing becomes feasible. The number of value ranges to consider is $b(b+1)/2 = O(b^2)$. An example of a histogram with four buckets and 10 possible $FVD(F_{new})$ is shown in Fig. 3.

Algorithm 6 Evaluate fragment F for any possible extract+migrates. R^m is the master replica of F and is currently located on site S_l.

```
refragment(F, R^m):
  fragmentations ← ∅
  for all S_r ∈ getActiveSites(R^m) do
    for all b_min ∈ H_cur[S_r, R^m], b_max ∈ H_cur[S_r, R^m] do
      card_rw ← histogramWriteCount(S_r, R^m, b_min, b_max)
      card_lw ← histogramWriteCount(S_l, R^m, b_min, b_max)
      utility ← w_BE · card_rw − card_lw − w_FS · card(F)
      if utility > 0 and (max − min + 1) > fragmentMinSize then
        fragmentations ← fragmentations ∪ (S_r, min, max, utility)
      end if
    end for
  end for
  sort(fragmentations) {Sort on utility value}
  removeIncompatible(fragmentations)
  for all (S_r, min, max, utility) ∈ fragmentations do
    F_1, F_new, F_2 ← extractNewFragment(F, min, max)
    migrateFragment(F_new, S_r) {Migrates master replica}
    updateReplicas()
  end for
  coalesceLocalFragments()
  histogramReorganize(R^m)
```

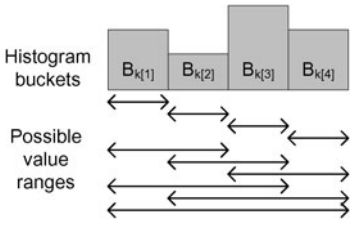


Fig. 3 Histogram with four buckets and corresponding value ranges

After the algorithm has completed, any adjacent fragments that now has master replicas on the same site are coalesced (denoted *coalesceLocalFragments*() in the algorithm). This helps keeping the number of fragments low. If two fragments are coalesced, the read replicas of those fragments must be updated as well. Some sites will likely have read replicas of only one of the fragments. These sites must either delete their replicas or get a replica of the fragment they are missing so coalescing can be performed on all replicas. Our heuristic is that we send the fragment which requires least communication cost to the sites missing that fragment. The remaining sites delete their local replicas.

Finally, old access statistics are removed from any remaining local master replicas using function *histogramReorganize*, as described in Sect. 5.2.4.

6.4 Cost functions

The core of the algorithms are the cost functions. The functions estimate the communication cost difference (or *utility*) between taking a given action (create, delete, split) and keeping the status quo. The basic assumption is that future accesses will resemble recent history as recorded in the access statistics histograms.

From Sect. 3.3 the communication cost $C_{total} = \sum_i C(A_i) + \sum_j C(RE_j)$. Accesses can either be reads, writes or updates: $\sum_i C(A_i) = \sum_k C(AR_k) + \sum_l C(AW_l) + \sum_m C(AU_m)$. The recent history for fragment F consists of a series of accesses $SA = [A_1, \ldots, A_n]$. Each access A_i comes from a site S_o. The accesses from a given site S_o is $SA(S_o)$ where $SA(S_o) \subset SA$. Since we measure communication cost, local accesses have no cost, i.e., $\forall A_i, A_i \in SA(S_l) \Rightarrow C(A_i) = 0$.

The basic form of the cost functions is as follow:

$$utility = benefit - cost \tag{1}$$

Replica creation: The benefit of creating a new read replica on site S_r is that reads from site S_r will become local operations and thus have no network communication cost. The cost of creating a new replica is first that the new replica will have to be updated whenever the master replica is written to. The second part of the cost is the actual transfer of the replica to the new site. This gives the following utility function:

$$utilityCreate = card(SR(S_r)) - card(SU) - card(F) \tag{2}$$

where $card(SR(S_r))$ is the number of reads from remote site S_r, $card(SU)$ is the number of replica updates and $card(F)$ is the size of the fragment.

Replica deletion: When a read replica R at site S_l is deleted, the benefit is that replica updates will no longer have to be transmitted to S_l. The cost is that local reads from S_l to R will now become remote. Thus we get the following utility function:

$$utilityDelete = card(SU) - card(SR(S_l)) \tag{3}$$

Splitting fragments and migrating master replicas: As described earlier, the algorithm handles splitting by using the cost function on all possible value ranges for the fragment. Thus the aim of the cost function is limited to estimating when a master replica R should be migrated from S_l to a remote site S_r. The only way a migration of the master replica can affect the number of remote reads and updates in the system, is if S_r already has a read replica. However, since S_l does not know the usage statistics of any possible replica at S_r, we simplify the function by omitting this possibility. The benefit of a migration of the master replica to S_r is therefore that writes from S_r will become local operations. Similarly, the cost will be writes from S_l. In addition we must consider the cost of migrating in itself. Our utility function:

$$utilityMigrate = card(SW(S_r)) - card(SW(S_l)) - card(F) \tag{4}$$

Cost function weights: While these equations are expressions of possible communication cost savings from different actions, they cannot be used quite as they are in

an actual implementation. There are a couple of issues. First, SW, SR and SU by design include only the recent history and cardinality values are therefore dependent on how much history we include. On the other hand, $card(F)$ is simply the current number of tuples in the fragment and thus independent on history size. We therefore scale $card(F)$ by a *cost function weight* w_{FS}. This weight will have to be experimentally determined and optimal value will depend on how much the usage history includes.

The second problem is stability. If we allow, e.g., migration when the number of remote accesses is just a few more than the number of local accesses, we could get an unstable situation where a fragment is migrated back and forth between sites. This is something we want to prevent as migrations cause delays in table accesses and indices may have to be recreated every time. To alleviate this problem, we scale the *benefit* part of the cost functions by $w_{BE} \in [0..1]$. For migrations, $w_{BE} = 0.5$ means that there will have to be 50% more remote accesses than local accesses for migration to be considered, i.e., for the utility to be positive (disregarding fragment size).

By including w_{FS} and w_{BE} we get the following cost functions:

$$utilityCreate = w_{BE} \cdot card(SR(S_r)) - card(SU) - w_{FS} \cdot card(F) \quad (5)$$

$$utilityDelete = w_{BE} \cdot card(SU) - card(SR(S_l)) \quad (6)$$

$$utilityMigrate = w_{BE} \cdot card(SW(S_r)) - card(SW(S_l)) - w_{FS} \cdot card(F) \quad (7)$$

Different values for the two cost function weights are evaluated experimentally in the Evaluation Section below.

7 Evaluation

In this section we present an evaluation of our approach. We aim to investigate different dynamic workloads and the communication cost savings our algorithms can achieve. Ideally, we would have liked to do a comparative evaluation with related work. However, to the best of our knowledge, no previous work exists that do continuous dynamic refragmentation and replication based on reads and writes in a distributed setting. Instead, we compare our results with a no-fragmentation and an optimal fragmentation method (where applicable).

The evaluation has three parts. First we examine the results from running a simulator on four workloads involving just two sites. These workloads have been designed to highlight different aspects, such as fragmentation, replication and changing access patterns. We have kept them as simple as possible to make it easier to analyze the results qualitatively. For the second part of the evaluation, we do simulations using two highly dynamic workloads involving more sites, providing a more realistic setting. The third part consists of experiments on an implementation in a distributed database system.

7.1 Experimental setup

For the evaluation, we implemented a simulator which allows us to generate distributed workloads, i.e., simulate several sites all performing tuple reads and writes

with separate access patterns. In all presented simulation results, the fragmentation and replication decision algorithms were run every 30 seconds. All simulations have been run 100 times, and we present the average values. For each simulated site, the following parameters can be adjusted:

- Fragmentation attribute value interval: minimum and maximum values for the accesses from the site.
- Access distribution: either uniform or hot spot (10% of the values get 90% of the accesses).
- Average rate of tuple accesses in number of accesses per minute. We use a Poisson distribution to generate accesses according to the frequency rate.
- Access type: reads, writes or a combination of both.

Values for these parameters need not be constant, but can change at any point for any site in the workload. In our simulations, we have used maximum histogram size of $MAX_B = 100$ buckets, and each table has one fragment with no read replicas when a simulation starts. We also tested with 1000 buckets, but this provided negligible benefits for our workloads.

Unlike most of the relevant previous work, our method tightly integrates fragmentation allocation and replication. Therefore, it does not make much sense comparing against techniques that only perform one of the tasks. Instead, we use the following two fragmentations methods to act as baselines for comparison. The first is a baseline where *the table is not fragmented or replicated at all*. The table consists of a single fragment with its master replica permanently allocated to the site with the largest total number of accesses. This is what would happen in a database system that does not use fragmentation or replication (e.g., to simplify implementation and configuration), at least given that workloads were completely predictable. Since there is no replication, there are no communication costs from migrations either.

The second allocation method we compare against, is *optimal fragmentation*. Here we assume full knowledge about future accesses. Each table is (at runtime) fragmented and the fragments are migrated and/or replicated to the sites which would minimize remote accesses.

It should be noted that both these fragment allocation alternatives assume advance knowledge about the fragmentation attribute value interval, distribution, frequency and type of accesses, none of which are required for our dynamic approach.

7.2 Workloads involving two sites

In this section, we present results from four workloads, each with two sites (S_1, S_2). These two sites accessed 25000–50000 tuples each. Early testing showed that 25000 tuples was more than enough to reach a stable situation. Only two sites were used for these workloads in order to make it easier to analyze the results. Each workload was therefore designed with a specific purpose in mind.

The fragmentation attribute value intervals for the two sites were designed so that they overlapped completely. Two rates were used, a high rate of 6000 accesses per minute and a low rate of 3000 accesses per minute. For workload 1 and 3, the workload was constant for both sites, while 2 and 4 switched workload parameters halfway

Table 4 Two-site workloads

Workload no.	Access	Distribution	Rate	Purpose
1	Write	S_1:Uniform, S_2:Hot spot	Low	Detect hot spots
2, first half	Write	S_1:Hot spot, S_2:Uniform	Low	Detect distribution change
2, second half	Write	S_1:Uniform, S_2:Hot spot	Low	
3	S_1:Read, S_2:Write	Uniform	S_1:High, S_2:Low	Make read replica
4, first half	S_1:Read, S_2:Write	Uniform	S_1:High, S_2:Low	Change replica pattern
4, second half	S_1:Write, S_2:Read	Uniform	S_1:Low, S_2:High	

through. Workloads 2 and 4 serve as examples of dynamic workloads where access patterns are not constant and predictable. The results from these workloads should illustrate if our approach's ability to adjust fragmentation and replication at runtime result in communication cost savings. The four workloads are detailed in Table 4.

Workload 1: In this workload, one of the sites has 10 hot spots while the other has uniform access distribution. Ideally, these 10 hot spots should be detected and migrated while the remainder should be left on the uniform access site. This case is similar to the one presented in Fig. 1. Figure 4(a) shows results for workload 1 with different values for w_{BE} and w_{FS}. Communication costs for *no-fragmentation* and *optimal fragmentation* are also shown.

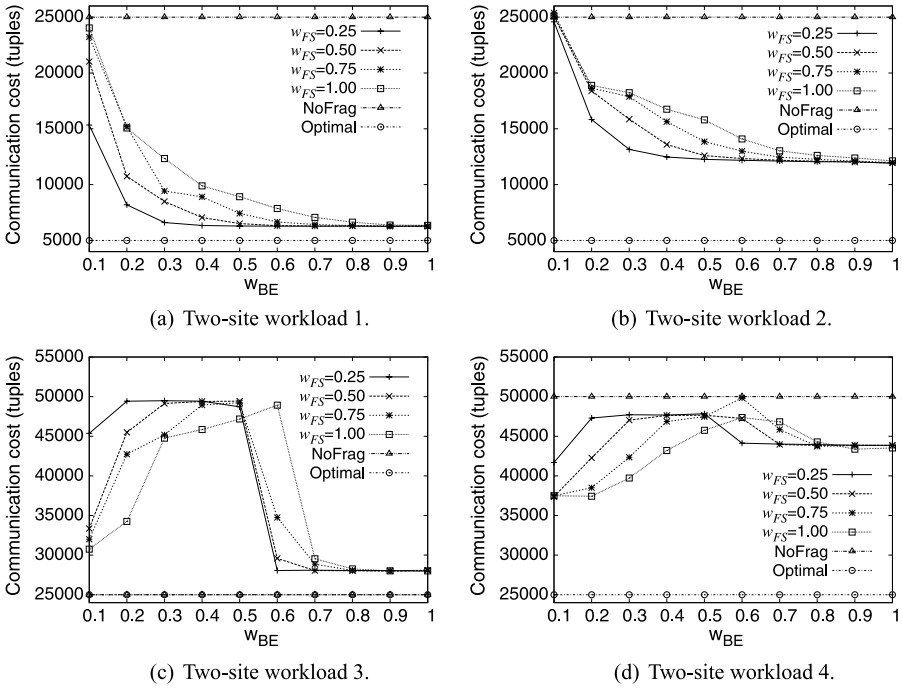
For this workload, the majority of the communication cost comes from remote writes, i.e. when the extract+migrate algorithm is very conservative on migrating the hotspots from S_1 to S_2. High values of w_{FS} cause the algorithm to overestimate the cost of migration while low values of w_{BE} cause the benefit to be undervalued. This combination thus almost reduces to the no-fragmentation case. For lower values of w_{FS} and higher values of w_{BE}, refragmentation decisions are made earlier and the result is comparable to optimal fragmentation.

Workload 2: This is a dynamic version of workload 1, with the two sites switching access patterns halfway through. The simulation results for this workload are shown in Fig. 4(b).

Results here are similar to workload 1, but with an extra overhead from detecting the access pattern change. This overhead is larger than for workload 1 simply because at the time the workload changes, the recent history is filled with the old workload and it takes a while for the new workload to dominate. The worst result is again similar to no-fragmentation.

Workload 3: This workload has one site writing while the other site reads at twice the rate. Ideally the site that writes should get the master replica, while the other site gets a read replica. Results from workload 3 are shown in Fig. 4(c).

The most important factor for the communication cost of this workload is whether a read replica is created on S_2. For low values of w_{BE}, the benefit of such a replica is undervalued and it is never created leading to poor results. Changes in w_{FS} only delay replica creation slightly and therefore has comparatively little influence. The exception is where high w_{FS} and low w_{BE} together prevent any migrations from happening, giving similar results to no-fragmentation. No-fragmentation does quite well



(a) Two-site workload 1. (b) Two-site workload 2.

(c) Two-site workload 3. (d) Two-site workload 4.

Fig. 4 (**a**) Results from two-site workload 1. (**b**) Results from two-site workload 2. (**c**) Results from two-site workload 3. Note that NoFrag and Optimal are equal for this workload, at 25000 tuples. (**d**) Results from two-site workload 4

here as it allocates the fragment to the site with the highest number of accesses which is also the optimal solution.

Workload 4: Similar to workload 3, except the two sites change behavior halfway through the workload. What we would like to see is a deletion of the read replica, migration of the master replica and a subsequent creation of a new read replica. Results from workload 4 are shown in Fig. 4(d).

The results are somewhat similar to workload 3. The largest difference is the overhead from detecting the workload change (similar to that of workload 2). For low values of w_{BE}, remote reads are the dominant cost since no replica is created. For higher values, a replica is created and remote updates dominates. No-fragmentation is now much worse since it does not adjust to the change.

Detailed results for all four workloads with $w_{BE} = 0.9$, $w_{FS} = 0.50$ are shown in Table 5. This table lists the number of remote accesses, migrations, fragments at the end of the run and the number of tuples transferred during migrations. The communication cost is the sum of remote accesses and tuples transferred. The final two columns shows the communication cost from the no-fragmentation and optimal allocation methods. Average results for the four workloads using the same cost function weight values are shown in Fig. 5.

Table 5 Detailed results, $w_{BE} = 0.9$, $w_{FS} = 0.50$

Workload no.	Re. writes	Re. reads	Re. updates	Migrations	Fragments	Tuples	Comm. cost	No frag.	Optimal
1	6229	0	0	10	20	46	6275	25000	5000
2	11145	0	0	53	47	860	12005	25000	5000
3	984	3154	22476	2	1	1385	27999	25000	25000
4	4009	6374	30310	44	21	3173	43866	50000	25000

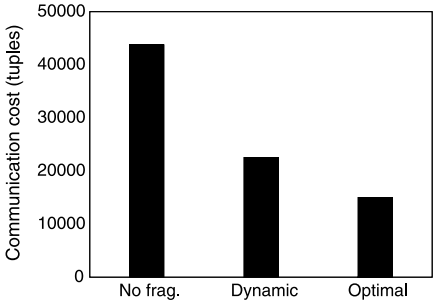


Fig. 5 Comparative results from two-site workloads

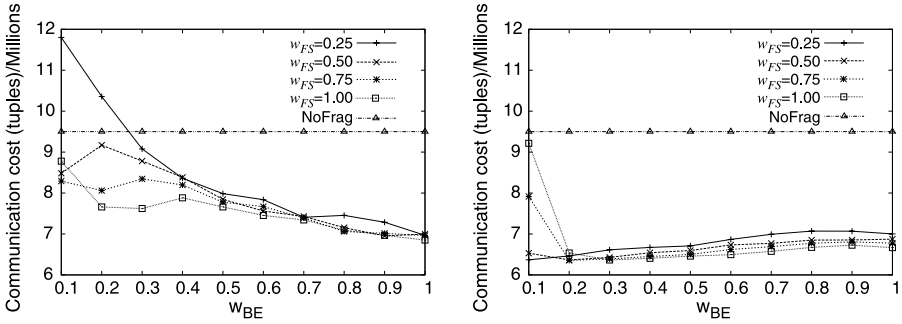
7.3 Workloads involving several sites

This section presents the results from two workloads involving 20 *active* sites each (i.e., the actual system can consist of a much larger number of sites, however only 20 sites simultaneously access the actual table during the simulation). The first of these workloads is intended to resemble a distributed application which have separate read and write phases, e.g., a grid application.

We have modeled the read phase as follows: A site uniformly accesses an random interval that constitutes 10% of the table. Between 30,000 and 60,000 reads are performed at an access rate of 2000 to 4000 reads a minute. Values for the interval, number of reads and rate are drawn randomly at the start of each phase.

After the read phase has completed, a write phase follows. Here the site accesses uniformly accesses a random interval 1% of the size of the table. Anywhere from 20,000 to 40,000 tuples are written at a rate of 2000 writes a minute. After the write phase has completed, a new read phase is initiated (and so on) until the site has accessed 500,000 tuples. With 20 sites, this gives a complete workload consisting of 10 million accesses. Also note that due to the random parameters, two different sites will generally not be in the same phase.

Comparative evaluation is more difficult for this workload than for those previously presented. The no-fragmentation method is still usable, but less realistic as the fixed non-fragmented master replica easily can become a bottleneck for remote writes and updates. The optimal fragmentation method is more problematic. With 10 million accesses each run and no clear access pattern, a very large number of fragmentations, migrations and replica allocations would have to be evaluated to find the optimal dynamic solution. Further, the highly random nature of this workload means that a



(a) Grid application workload. (b) General workload

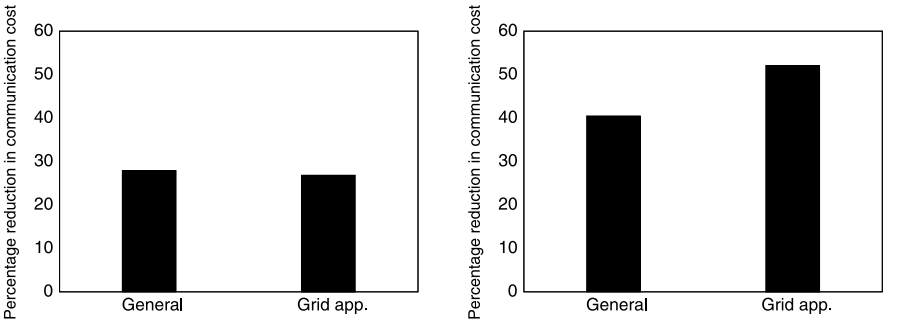
Fig. 6 (**a**) Results from grid application workload. (**b**) Results from general workload

fragmentation and replica allocation that are optimal for one run, will not be optimal for another. The optimal fragmentation method would therefore have to be recomputed for each run. For these reasons we found optimal fragmentation infeasible and omitted it from this part of the evaluation.

The results are shown in Fig. 6(a). With ten times as many sites as for earlier workloads, having too many replicas becomes a much more important issue due to the number of update messages needed to keep all the replicas consistent. This is what causes very poor results with a combination of low w_{BE} and low w_{FS}. The low w_{FS} underestimates the cost of creating a read replica while low w_{BE} makes it hard to delete it later. This leads to an excessive number of replicas and poor performance from the high number of updates needed. Due to the highly dynamic nature of this workload, high values of w_{BE} work well as they make the algorithms take action earlier. Since the number of writes is low and confined to narrow intervals of the table, fragment sizes stay small and thus the w_{FS} value is of little importance.

The second multi-site workload is intended to resemble a more general usage pattern where each site does not have distinct read and write phases, but rather a single phase that includes both. We have modeled it as follows: A site uniformly accesses a random interval that constitutes 10% of the table. Each of these accesses can be either a read (80%) or a write (20%). The access rate is from 2000 to 4000 accesses a minute, and the phase lasts between 30,000 and 60,000 accesses. After the phase has completed, it restarts with new sets of parameters randomly drawn. As for the last workload, this continues until 500,000 accesses have been made from each site. The simulation results are shown in Fig. 6(b).

Similar to the grid application workload, the creation and deletion of read replicas are the most important factors influencing the results. Low values of w_{BE} make the algorithms act conservatively, both when creating and deleting replicas. This leads to remote reads dominating the communication cost. For higher values of w_{BE}, more replicas are created giving fewer remote reads but more updates. For this workload, these two factors tended to balance each other out, giving similar communication costs for a wide selection of cost function weight values. While there are separate write phases in the grid application workload that each only accessed 1% of the table, writes in this workload were interleaved with reads and accessed a much larger part



(a) Multi-site workloads in simulations. (b) Multi-site workloads in DASCOSA-DB.

Fig. 7 (**a**) Comparative results from simulations with multi-site workloads, showing reduction in communication cost relative to the no-fragmentation method. (**b**) Comparative results from multi-site workloads using DYFRAM implemented in DASCOSA-DB, showing reduction in communication cost relative to the no-fragmentation method

Table 6 Tuples transferred during multi-site workloads in simulations and implementation in DASCOSA-DB

	Simulation			Implementation		
Workload	No frag.	DYFRAM	Reduction	No frag.	DYFRAM	Reduction
General	9.5 mill.	6.85 mill.	27.9%	100,000	59519	40.5%
Grid app.	9.5 mill.	6.95 mill.	26.8%	100,000	47921	52.1%

of the table (for a given phase). This workload also had a smaller fraction of the accesses as writes. These three factors caused the splitting algorithm to create smaller fragments which meant that w_{FS} had little impact on the results.

Comparative results for the two multi-site workloads using $w_{BE} = 0.9$ and $w_{FS} = 0.50$, are shown in Fig. 7(a) and Table 6.

7.4 Implementation of DYFRAM in DASCOSA-DB

In this experiment, DYFRAM was implemented in the DASCOSA-DB distributed database system [16] in order to verify simulation results. The workloads tested are similar to the grid and general workloads presented in Sect. 7.3, but have been scaled down a bit for practical reasons.

The grid workload has read phases of 6,000–12,000 accesses. Each phase uniformly accesses a random 5% interval of the table. Write phases do 4,000–8,000 writes to a random 0.5% interval of the table. There is no delay between accesses. As soon as a site finishes one phase, it starts on the next, alternating between read and write phases. The experiments are done with 6 sites, each issuing 20,000 accesses, i.e., a total of 120,000 accesses. Half of the sites start in a read phase, while the other half starts in a write phase. Due to the different phase lengths, this pattern will change several times during the experiment.

The general workload is scaled with the same factors, giving phases of 6,000–12,000 accesses to 5% of the table. 80% of these are read accesses and 20% are write

accesses. Each of the 6 sites issues 20,000 accesses, resulting in a total of 120,000 accesses.

The refragmentation algorithm is run every 30 seconds with $w_{FS} = 0.2$ and $w_{BE} = 0.95$, which should give a quite aggressive use of refragmentation and replication. As explained in Sect. 6.4, the weights were found experimentally by testing on a shorter workload, consisting only of a few thousand accesses. The results are compared against the no-fragmentation method. Each experiment is repeated a number of times with different random seeds.

The results from both workloads and the no-fragmentation method are shown in Fig. 7(b) and Table 6. We see that the results are similar to those from the simulations. For the general workload, communication costs are reduced by more than 40% compared to the no-fragmentation method. The costs of the grid workload is reduced by more than 50%. Clearly, the cost of replication is made up for by converting remote accesses to local accesses. Around 20% of the tuples transferred are caused by fragments moving around. The ratio of read vs. write accesses varies more, with the grid workload generally having higher write costs and the general workload having higher read costs.

The results do not vary much between each run, and small changes in w_{FS} and w_{BE} do not change the results much. The length of each phase will affect the cost savings, but even if phases are only half as long, communication costs are 25% below the no-fragmentation method.

8 Conclusions and further work

In distributed database systems, tables are frequently fragmented and replicated over a number of sites in order to reduce network communication costs. How to fragment, when to replicate and how to allocate the fragments to the sites are challenging problems that has previously been solved either by static fragmentation and allocation, or based on the analysis of a priori known queries. In this paper we have presented *DYFRAM*, a decentralized approach for dynamic table fragmentation and allocation in distributed database systems, based on observation of the access patterns of sites to tables. To the best of our knowledge, no previous work exists that perform the combination of continuous refragmentation, reallocation, and replication in a distributed setting.

Results from simulations show that for typical workloads, our dynamic fragmentation approach significantly reduces communication costs. The approach also demonstrates well its ability to adapt to workload changes. In addition to simulations, we have also implemented DYFRAM in the DASCOSA-DB distributed database system, and demonstrated its applicability in real applications.

Future work include exploring adaptive adjustment of the cost function weights as well as better workload prediction based on control theoretical techniques. We also intend to develop a variant of our approach that can be used in combination with static query analysis in order to detect periodically recurring access patterns.

References

1. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of SIGMOD, 2004
2. Agrawal, S., Chu, E., Narasayya, V.R.: Automatic physical design tuning: workload as a sequence. In: Proceedings of SIGMOD 2006, 2006
3. Ahmad, I., et al.: Evolutionary algorithms for allocating data in distributed database systems. Distrib. Parallel Databases **11**(1), 5–32 (2002)
4. Apers, P.M.G.: Data allocation in distributed database systems. ACM Trans. Database Syst. **13**(3), 263–304 (1988)
5. Bonvin, N., Papaioannou, T.G., Aberer, K.: A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In: Proceedings of SoCC '10, 2010
6. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: Proceedings of ICDE, 2007
7. Brunstrom, A., Leutenegger, S.T., Simha, R.: Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In: Proceedings of CIKM '95, 1995
8. Ciciani, B., Dias, D., Yu, P.: Analysis of replication in distributed database systems. IEEE Trans. Knowl. Data Eng. **2**(2), 247–261 (1990)
9. Copeland, G., et al.: Data placement in Bubba. In: Proceedings of SIGMOD 1988, 1988
10. Corcoran, A.L., Hale, J.: A genetic algorithm for fragment allocation in a distributed database system. In: Proceedings of SAC'94, 1994
11. Didriksen, T., Galindo-Legaria, C.A., Dahle, E.: Database de-centralization—a practical approach. In: Proceedings of VLDB 1995, 1995
12. Donjerkovic, D., Ioannidis, Y.E., Ramakrishnan, R.: Dynamic histograms: Capturing evolving data sets. In: Proceedings of ICDE, 2000
13. Furtado, P.: Experimental evidence on partitioning in parallel data warehouses. In: Proceedings of DOLAP 2004, 2004
14. Gavish, B., Sheng, O.R.L.: Dynamic file migration in distributed computer systems. Commun. ACM **33**(2), 177–189 (1990)
15. Hara, T., Madria, S.K.: Data replication for improving data accessibility in ad hoc networks. IEEE Trans. Mob. Comput. **5**(11), 1515–1532 (2006)
16. Hauglid, J.O., Nørvåg, K., Ryeng, N.H.: Efficient and robust database support for data-intensive applications in dynamic environments. In: Proceedings of ICDE, 2009
17. Hua, K.A., Lee, C.: An adaptive data placement scheme for parallel database computer systems. In: Proceedings of VLDB 1990, 1990
18. Ioannidis, Y.: The history of histograms (abridged). In: Proceedings of VLDB 2003, 2003
19. Ivanova, M., Kersten, M.L., Nes, N.: Adaptive segmentation for scientific databases. In: Proceedings of ICDE 2008, 2008
20. Menon, S.: Allocating fragments in distributed databases. IEEE Trans. Parallel Distrib. Syst. **16**(7), 577–585 (2005)
21. Mondal, A., Madria, S.K., Kitsuregawa, M.: CADRE: A collaborative replica allocation and deallocation approach for mobile-p2p networks. In: Proceedings of IDEAS 2006, 2006
22. Mondal, A., Yadav, K., Madria, S.K.: EcoBroker: An economic incentive-based brokerage model for efficiently handling multiple-item queries to improve data availability via replication in mobile-p2p networks. In: Proceedings of DNIS 2010, 2010
23. Padmanabhan, P., Gruenwald, L., Vallur, A., Atiquzzaman, M.: A survey of data replication techniques for mobile ad hoc network databases. VLDB J. **17**(5), 1143–1164 (2008)
24. Rao, J., et al.: Automating physical database design in a parallel database. In: Proceedings of SIGMOD 2002, 2002
25. Saccà, D., Wiederhold, G.: Database partitioning in a cluster of processors. ACM Trans. Database Syst. **10**(1), 29–56 (1985)
26. Shin, D.-G., Irani, K.B.: Fragmenting relations horizontally using a knowledge-based approach. IEEE Trans. Softw. Eng. **17**(9), 872–883 (1991)
27. Sidell, J., Aoki, P.M., Sah, A., Staelin, C., Stonebraker, M., Yu, A.: Data replication in Mariposa. In: Proceedings of ICDE 1996, 1996
28. Stonebraker, M., et al.: Mariposa: A wide-area distributed database system. VLDB J. **5**(1), 48–63 (1996)
29. Tamhankar, A., Ram, S.: Database fragmentation and allocation: an integrated methodology and case study. IEEE Trans. Syst. Man Cybern., Part A **28**(3), 288–305 (1998)

30. Ulus, T., Uysal, M.: Heuristic approach to dynamic data allocation in distributed database systems. Pak. J. Inf. Technol. **2**(3), 231–239 (2003)
31. Weikum, G., et al.: The COMFORT automatic tuning project, invited project review. Inf. Syst. **19**(5), 381–432 (1994)
32. Wolfson, O., Jajodia, S.: Distributed algorithms for dynamic replication of data. In: Proceedings of PODS'92, New York, NY, USA, 1992. ACM, New York (1992)
33. Wong, E., Katz, R.H.: Distributing a database for parallelism. SIGMOD Rec. **13**(4), 23–29 (1983)
34. Zilio, D.C., et al.: DB2 design advisor: integrated automatic physical database design. In: Proceedings of VLDB 2004, 2004