Norwegian University of Science and Technology
Technical Report IDI-TR-02/2008

DYTAF: Dynamic Table Fragmentation in Distributed Database Systems

Jon Olav Hauglid, Kjetil Nørvåg and Norvald H. Ryeng
DASCOSA Group, Dept. of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
E-mail: {joh,noervaag,ryeng}@idi.ntnu.no

ISSN: 1503-416X

Abstract

In distributed database systems, tables are frequently fragmented over a number of sites in order to reduce access costs. How to fragment and how to allocate the fragments to the sites is a challenging problem that has previously been solved either by static fragmentation and allocation, or based on query analysis. Many emerging applications of distributed database systems generate very dynamic workloads with frequent changes in access patterns from different sites. In those contexts, continuous refragmentation and reallocation can significantly improve performance. In this paper we present DYTAF, 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 and reallocation based on recent access history, aiming at being able to maximize the number local accesses compared to accesses from remote sites. Through simulations we show that the approach gives close to optimal performance for typical access patterns and thus demonstrate the feasibility of our approach.

1 Introduction

There is an emerging need for efficient support for databases consisting of very large amounts of data that are created and used by sites at different physical locations. Examples of application areas include Grid databases, distributed data warehouses, and large distributed enterprise databases.

In distributed databases, communication cost can be reduced by partitioning database tables horizontally into *fragments*, and allocate these fragments to the sites where they are most frequently accessed. The aim is to make most data accesses local, and avoid remote read/write. Obviously the big challenges are *how to fragment* and *how to allocate*.

In many of the application areas above there is a very dynamic workload with frequent changes in access patterns of different sites. One common reason for this is that data usage often consists of two separate phases: a first phase where writing of data dominates (for example during simulation when results are written), and a subsequent second phase when data is mostly read. The dynamism of the total access pattern is further increased by different instances of the applications being in different phases.

Previous work on data allocation has focused on (mostly static) fragmentation based on queries, which is appropriate for the second phase. However, these techniques are most useful in contexts where read queries dominate and where decisions can be made based on SQL-statement analysis. In general these techniques also involve a centralized computation based on collected statistics from participating sites.

Because of dynamic workloads, static/manual fragmentation can not be optimal. Instead, the fragmentation and fragment allocation should be dynamic and completely automatic, i.e., changing access patterns should result in refragmentation and reallocation of fragments when beneficial.

In this paper we present *DYTAF*, 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 and reallocation based on recent access history, aiming at being able to maximize the number local accesses compared to accesses from remote sites.

An example of what we aim at with our approach is illustrated in Fig. 1, where the figure illustrates the access pattern to 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 be 6 fragments, one for each of the hot spot areas and one for each of the areas between. 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

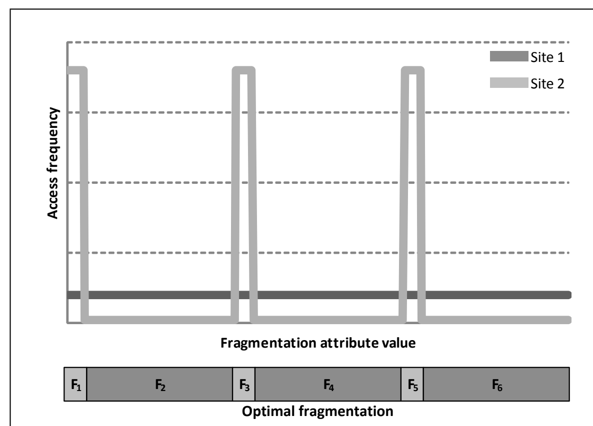

Figure 1: Example access pattern, and desired fragmentation and allocation.

the experimental evaluation, DYTAF will detect this pattern, split the table into the appropriate fragments, and then allocate these fragments to the appropriate sites. Note that if the access pattern changes later, this will be detected and fragments reallocated in addition to possible repartitioning.

The main contributions of this paper are 1) a low-cost algorithm for fragmentation decisions, making it possible to perform refragmentation based on the current workload, and 2) dynamic reallocation of fragments in order to minimize total access cost in the system. The process is performed completely decentralized, i.e., without a particular controlling site. An important aspect of our approach is *the combination of the refragmentation and reallocation process.* To our knowledge, no previous work exists that do dynamic refragmentation based on both reads and writes in a distributed setting.

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. In Section 3 we outline the assumed system and fragment model and state the problem tackled in this work. In Section 4 we give an overview of DYTAF. In Section 5 we describe how to manage fragment access statistics. In Section 6 we describe in detail the dynamic table fragmentation algorithm. In Section 7 we evaluate the usefulness of our approach. Finally, in Section 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 [4, 10, 11, 16, 18, 22]. Given a set of common queries, these methods

describe a fragmentation that is optimized for the query load. The problems with these are that they focus on only queries [11, 16] or static placement of fragments when the query set is known [4, 10, 18, 22]. Some methods also use more particular information on the data in addition to the query set [19]. This information is provided by the user, and is not available in a fully automated system.

Our approach does not look at only the queries, but a combination of queries, inserts, updates and deletes. It can be argued that the workload should be viewed as a sequence of operations, not a set [1], which is the approach we have taken. Also, our solution handles both dynamic fragmentation and allocation.

Fragmentation, together with other physical database tuning, such as creating and dropping indices and materialized views, can be viewed as something to do only when doing major reconfiguration [17], possibly with the aid of a design advisor that suggests possible courses of action [23]. On the other hand, fully automatic tuning [21] 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 [2].

Using our method, fragments are automatically split, coalesced and reallocated 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. [6], 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.

Mariposa [20] is a notable exception to the traditional, manually fragmented systems. Mariposa provides refragmentation and reallocation 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 complex 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 DYTAF site will split off or reallocate a fragment if it optimizes access to the fragment, seen from the fragment's viewpoint. This is done also during query execution, not only as part of query planning, as is the case in Mariposa.

Adaptive indexing [1, 5] 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 [7, 13], or on query analysis [15]. In our algorithms we seek to place data on the sites where they are being used, whether it is reading or writing, not to balance the load.

During the last years, a number of papers exploring the use of evolutionary algorithms in the fragment allocation task has appeared [3, 8]. However, they do not look into how the tables should be fragmented, and not how to cope with dynamic workloads.

Obviously some of the research in distributed file systems (see summary in [12]) is also relevant to our approach. One important difference between distributed file systems and distributed database systems is the typical granularity of data under consideration (files vs. tuples) and the availability of a fragmentation attribute that can be used for partitioning in distributed database systems. Our system adapts its granularity, i.e., fragment size, to the current workload.

Our approach is to refragment on the fly, based on current operations and recent history of reads and writes. Refragmentation is done automatically without user interaction. 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 to make fragmentation decisions.

3 Preliminaries

In this section we present preliminaries that provides the context for the rest of the paper. We also introduce symbols to be used throughout the paper, summarized in Table 1.

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 capabilities and communication capacities.

Each site has a DBMS, and a site can access local data and take part in the execution of distributed queries, i.e., the DBMSs together constitute a distributed database system. The distribution aspects can be supported directly by the local DBMS or can be provided through middleware.

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 both 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 afterwards propagated to the replicas. Beyond this, replication has little impact on our approach. If a master replica gets refragmented, other replicas must be notified so they can be refragmented as well. Beyond the notification, this should not incur any extra communication cost.

3.2 Fragment model

Fragmentation is based on one attribute value having a domain D, and each fragment covers 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 $\cup$*FVD*$(F_i) = [D_{min}, D_{max}]$ A fragment F_{prev} can subsequently be split into two or more fragments $F_1, \ldots, F_n$. In this case, the following is true:

$$\cup_{i=1}^{n} F_i = F_{prev}$$

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

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:

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

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

Symbol	Description
S_i	Site
t_i	Tuple
T	Table T
F_i	Fragment i of table T
$F_i[min, max]$	Fragment value domain
$\mathcal{F}_t$	Fragmentation at time t
C	Cost
A_i	Tuple access
RE_j	Refragmentation

Table 1: Symbols.

Assume a distributed database system consisting of a number of sites $S_i, i = 1 \ldots n$ and a global table T. At any time t the table T has a certain fragmentation $\mathcal{F}_t = \{S_0(F_0, F3), S_3(F_1, F_2)\}$. Note that not all sites have allocated fragments, and that there in addition might be replicas of fragments created based on read pattern.

3.3 Problem definition

During operation, tuples are accessed as part of read or write operations. If the fragment where a tuple belong (based on the value of the fragmentation attribute) is stored on the same site as the site S_a performing the access A, the cost is $C(A) = C_L$. On the other hand, if the fragment is stored on a remote site, a remote access has to be performed, which has a cost of $C(A) = C_R$. In this paper we focus on reducing the communication costs, and let $C_L = 0$. 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 $[A_1, ..., A_n]$. The total access cost is $C = \sum_i C(A_i)$. The access cost of a tuple at a particular time instant depends on the fragmentation $\mathcal{F}_t$.

Refragmentation and reallocation of fragments can be performed at any time. The main cost of refragmentation (given a computationally cheap algorithm for determining fragmentation and allocation) is the migration of fragments from one site to another. We denote the total cost of one refragmentation/reallocation as $C = C(RE_j)$ (this includes any regeneration of indices after migration). Thus the total refragmentation/reallocation during the time the system is observed is $C = \sum_j C(RE_j)$.

The combined cost of access and refragmentation/reallocation is thus $C_{total} =$

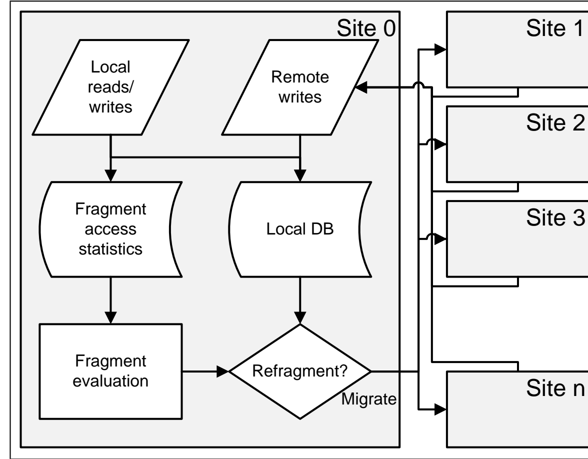

Figure 2: Dynamic fragmentation and allocation.

$\sum_i C(A_i) + \sum_j C(RE_j)$. Note that the access and refragmentation/reallocation operations are interleaved. The aim of our approach is to optimize the cost C_{total}.

4 Overview of DYTAF

This section describes our approach to dynamically fragment tables and allocate those fragments to different sites in order to improve locality of table accesses and thus reduce communication costs. Each site makes the decision to split and/or migrate its local fragments independently of other sites. In case of replication, only the master replicas are considered. This means that no extra communication between sites is necessary to make fragmentation decisions. Our approach has two main parts: 1) detecting fragment access patterns, and based on these statistics to 2) decide on refragmentation and reallocation. The approach is illustrated in Fig. 2.

In order to make informed decisions about useful fragmentation 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 fragment access patterns, i.e., which sites are accessing which parts of fragments. This is performed by recording fragment accesses in order to discover access patterns. Recording accesses is performed continuously, with old data periodically discarded to make the 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 Section 5.

Given available statistics, our table fragmentation algorithm examines accesses

for each fragment and evaluates possible refragmentations and reallocations based on recent history. The algorithm runs at given intervals, individually for each fragment. A vital component of this algorithm is a cost function that estimates the difference in future communication costs between a given refragmentation and keeping the fragment as is. Details are presented in Section 6.

5 Fragment access statistics

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

Since we only record accesses to the master replica, accesses will not include reads performed on non-master replicas. This choice was made for two reasons: 1) we focus on optimizing writes as optimizing reads has been previously studied (see Section 2), and 2) recording accesses to the master replicate can be performed without involving sites with other replicas. In cases where recording all reads is important, our approach can easily be extended by recording statistics on all replica sites and transmit them periodically to the master replica site.

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 access 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. For this reason, we have selected to use a set of histograms, one for each site, as the basic data structure for each fragment. These histograms must be small enough to be kept in main memory for efficient processing.

Symbol	Description
H_i	Histogram
b_k	Histogram bucket number
$B_i[b_k]$	Histogram bucket
W	Bucket width
MAX_B	Maximum number of buckets
Z_W	Factor used when resizing buckets

Table 2: Histogram symbols.

Below 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 [14]. Most of these have been *static histograms* constructed from an existing data series and then left unchanged. In our case, data arrives and must be recorded in the histogram 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* [14, 9], that are maintained incrementally and therefore more 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 [14].

For our approach we have instead chosen to use equi-width histograms. This was done in order to improve the performance of histogram operations, since equi-width histograms by design are simpler to use and 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 equally well to equi-depth histograms. However, the considerably reduced computational cost makes this an acceptable trade-off.

Histogram-related symbols used in the following discussion are summarized in Table 2. The value of bucket number b_k of histogram H_i is denoted as $B_i[b_k]$. We use equi-width histograms with bucket width W and limit bucket value ranges to start and end on multiples of W. This means that the value range covered by

bucket $B_i[b_k]$ is $[b_k * W, (b_k + 1) * W)$.

Histograms only maintain statistics for values that are actually used, i.e., they do not cover the whole *FVD*. This saves space by not storing empty buckets, which is useful since we lack a priori knowledge about fragment attribute values. Buckets are therefore stored as $(b_k, B_i[b_k])$ pairs 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, if the number of buckets used is small, bucket width is decreased by the same factor. This is performed to make sure we have as many buckets as possible, as this better captures the fragment 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 algorithm has 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 fragmentation algorithm 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 fragment F_j as $H_{cur}[S_i, F_j]$, while the old histogram is $H_{old}[S_i, F_j]$

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 fragments is accessed, the corresponding histogram is updated. This is described in Algorithm 1. Although not included in the algorithms to improve clarity, we perform value normalization on values before they are entered into the histogram. Assuming a 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 $[b_k * W, (b_k + 1) * W)$. Processing is therefore determining b_k for a given fragmentation attribute value v_j and incrementing the bucket value. The formula is $b_k = v_j / W$ which means

Algorithm 1 Site S_i accesses tuple t_j in fragment F_j with fragmentation attribute value v_j.

```
histogramUpdate(S_i, F_j, v_j){
   H_i ← H_cur[S_i, F_j]
   b_k ← v_j/W
   B_i[b_k] ← B_i[b_k] + 1
   if numberOfBuckets > MAX_B then
      increaseBucketWidth(F_j)
   end if
}
```

processing is O(1). Also, since histograms are kept in main memory, histogram updates do not incur any disk accesses.

If no bucket already exist for bucket number b_k, a new bucket must be constructed. This is the only time 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 [14]. 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 $getActiveSites(F)$ returns the set of all sites that have written to fragment F.

Similarly, if we at any point only use a very low number of buckets, the bucket widths can be decreased in order to make access statistics more accurate. Algorithm 3 describes how this is done. Of special note is the expression $max(1, B_i[b_k]/Z_W)$.

Algorithm 2 Increase bucket width W for histograms for fragment F by factor Z_W.

$increaseBucketWidth(F)\{$
  for all $S_i \in getActiveSites(F)$ **do**
    for all $H_i \in H_{cur}[S_i, F] \cup H_{old}[S_i, F]$ **do**
      $H'_i \leftarrow 0$ $\{'$ indicates temporary values$\}$
      for all $B_i[b_k] \in H_i$ **do**
        $b'_k = b_k / Z_W$
        $B'_i[b'_k] = B'_i[b'_k] + B_i[b_k]$
      end for
      $H_i \leftarrow H'_i$
    end for
  end for
$\}$

If the large bucket to be divided into smaller buckets contain only a few tuples, rounding can make $B_i[b_k]/Z_W = 0$ 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.

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. Range count is shown in Algorithm 4. In order to get the sum of range counts from all sites, the function $histogramRangeCountAll(F, b_{min}, b_{max})$ is used.

5.2.4 Histogram reorganization

As stated earlier, it is important that only the recent access history is used for fragment evaluations in order to make it possible to adapt to changes in access patterns. This is done by having two sets of histograms. How these two sets are swapped is described in Algorithm 5.

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

Algorithm 3 Decrease bucket width W for histograms for fragment F by factor Z_W.

$decreaseBucketWidth(F)\{$
  for all $S_i \in getActiveSites(F)$ **do**
    for all $H_i \in H_{cur}[S_i, F] \cup H_{old}[S_i, F]$ **do**
      $H'_i \leftarrow 0$
      for all $B_i[b_k] \in H_i$ **do**
        for $b'_k = 0$ to Z_W **do**
          $B'_i[b_k * Z_W + b'_k] = max(1, B_i[b_k]/Z_W)$
        end for
      end for
      $H_i \leftarrow H'_i$
    end for
  end for
$\}$

Algorithm 4 Count number of accesses from site S to fragment F stored in buckets numbered $[b_{min}, b_{max}]$.

$histogramRangeCount(S, F, b_{min}, b_{max})\{$
  $n \leftarrow 0$
  $H_{cur} \leftarrow H_{cur}[S, F]$
  $H_{old} \leftarrow H_{old}[S, F]$
  for $b_i = b_{min}$ to b_{max} **do**
    $n \leftarrow n + B_{cur}[b_i] + B_{old}[b_i]$
  end for
  return n
$\}$

Algorithm 5 Performed periodically to remove old access history statistics for fragment F.

$histogramReorganize(F)\{$
  for all $S_i \in getActiveSites(F)$ **do**
    $H_{old}[S_i, F] \leftarrow H_{cur}[S_i, F]$
    $H_{cur}[S_i, F] \leftarrow 0$
  end for
  if $numberOfBuckets * Z_W < MAX_B$ **then**
    $decreaseBucketWidth(F)$
  end if
$\}$

still stay below the bucket number maximum MAX_B.

5.3 Histogram memory requirements

It is important that the size of the histograms is small so enough main memory is available for more efficient query processing and buffering. For every master replica a site has, it must store two histograms for each active site accessing the fragment. Every bucket is stored as a $(b_k, B_i[b_k])$ pair. Assuming b buckets and c active sites, the memory requirements for each fragment is $2*c*b*sizeOf(bucket)$ or $O(b * c)$. Since b have an upper bound MAX_B, memory consumption does not depend on fragment size or number of accesses, only the number of active sites.

6 Dynamic table fragmentation algorithm

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*).

More formally, assume a fragmentation $\mathcal{F}_{old}$ which includes a fragment F_i with *FVD*$(F_i) = F_i[min_i, max_i]$ allocated to site S_i. Find a set of fragments $F_m, ..., F_n$ such that $\cup F_m, ..., F_n = F_i$ with $F_{new} \in F_m, ..., F_n$ allocated to site $S_k \neq S_i$ such that communication cost $C_{total} = \sum C(A_i) + \sum C(RE_j)$ decreases.

Below we first present the algorithm itself before we describe the cost function used to predict future communication costs from a given refragmentation.

6.1 Algorithm details

The algorithm is run for each fragment a given site has the master replica of, at regular intervals. The result of each execution can be either: 1) do nothing, i.e, the fragment is where it should be, 2) migrate the whole fragment, or 3) extract a new fragment F_{new} with *FVD*$(F_{new}) = F_{new}[min_{new}, max_{new}]$ and migrate it. If any fragments are migrated, the algorithm is run again on the remaining fragments until no further changes are made. A decision to migrate the whole fragment 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}.

The algorithm for evaluating and refragmenting a given fragment F is presented in Algorithm 6. In the algorithm the cost function (to be described in detail below) is applied to all possible F_{new} for each remote site S_r that has written to F. The result is a *utility value* that estimates the communication cost reduction from extracting and migrating F_{new} to S_r. After all possible F_{new} and S_r have been evaluated, the extract+migrate with highest utility is executed. In the case of no alternatives giving a utility larger than 0, no refragmentation will be made. Note that no fragments with *FVD* less than *fragmentMinSize* will be extracted in order to prevent refragmentation to result 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 a new fragment F_{new} such that *FVD*$(F_{new}) = F_{new}[min_{new}, max_{new}]$ ∈*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 simplification is necessary.

The reduction in number of possible fragments to consider is obtained based on the following observation:

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

If any F_{new} is migrated, the algorithm runs recursively on any remaining fragments. The minimum size of *FVD*(F_{new}) is one bucket width or W. With respect to run time, the worst case is a new fragment of minimum size extracted from each recursive iteration. This means that the maximum recursive depth is equal to the

Algorithm 6 Evaluate fragment F for any possible extract+migrates. F is currently located on site S_l.

$dynamicFragmentation(F)\{$

$bestUtility \leftarrow 0$
for all $S_r \in getActiveSites(F)$ **do**
  $H_{cur} \leftarrow H_{cur}[S_r, F]$
  $H_{old} \leftarrow H_{old}[S_r, F]$
  for all $B_{cur[min]} \in H_{cur}, B_{old[min]} \in H_{old}$ **do**
    for all $B_{cur[max]} \in H_{cur}, B_{old[max]} \in H_{old}$ **do**
      $card_r \leftarrow histogramRangeCount(S_r, F, min, max)$
      $card_l \leftarrow histogramRangeCount(S_l, F, min, max)$
      $card_A \leftarrow histogramRangeCountAll(F, min, max)$
      $utility \leftarrow card_r - w_{Sl} * card_l - w_{SA} * card_A - w_F * card(F)$ {Explained in Section 6.2}
      if $utility > bestUtilty$ **and** $(max - min + 1) >$ *fragmentMinSize* **then**
        $bestutility \leftarrow utility$
        $S_{migrate} \leftarrow S_r$
        $bestMin \leftarrow min$
        $bestMax \leftarrow max$
      end if
    end for
  end for
end for
if $bestUtility > 0$ **then**
  $F_1, F_{new}, F_2 \leftarrow extract(F, bestMin, bestMax)$
  $migrateFragment(F_{new}, S_r)$ $dynamicFragmentation(F_1)$
  $dynamicFragmentation(F_2)$
else
  $coalesceLocalFragments(F)$
  $histogramReorganize(F)$
end if

$\}$

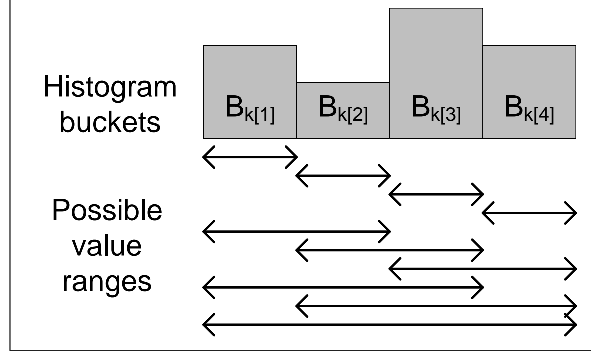

Figure 3: Histogram with four buckets and corresponding value ranges.

number of buckets.

Finally, after the algorithm has completed, any adjacent fragments that now reside on the same site, are coalesced (denoted $coalesceLocalFragments()$ in the algorithm). This helps keep the number of fragments down. Also, old access statistics are removed ($histogramReorganize()$) from any remaining local fragments as described in Algorithm 5.

6.2 Cost function

The core of the dynamic table fragmentation algorithm is the cost function. Assuming a fragment F is currently allocated to site S_l. The function estimates the communication cost difference between keeping F as is, and migrating it to S_r. The basic assumption is that future fragment accesses will resemble recent history as recorded in the access statistics histograms.

From Section 3.3 we have that the communication cost is $C_{total} = \sum_i C(A_i) + \sum_j C(RE_j)$. We first consider keeping the fragment F on site S_l. This means no migration, i.e., $\sum_j C(RE_j) = 0$. The recent history for fragment F consists of a series of accesses $SA = [A_1, ..., A_n]$. Each access A_i comes from a site S_k. The accesses from a given site S_k is $SA(S_k)$ where $SA(S_k) \subset SA$. 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 communication cost for one remote access is C_R, thus:

$$C_{keep} = C_R * card(SA - SA(S_l)) \tag{1}$$

where $card(SA)$ is the number of accesses.

Alternatively we can migrate F to S_r with the communication cost $C(RE_j)$. After the migration is complete, accesses from S_r will become local (i.e., no cost),

while accesses from S_l will become remote and now cost C_R each. The communication cost is therefore:

$$C_{migrate} = C(RE_j) + C_R * card(SA - SA(S_r)) \quad (2)$$

Our basic assumption is that a migration should be made if recent history indicates that the fragment should have been migrated a while ago, i.e. fragment F should be migrated if $C_{migrate} < C_{keep}$. Since accesses from sites other than S_l and S_r will be remote accesses no matter what, their cost is irrelevant and can be omitted. We should therefore migrate F to S_r if:

$$C(RE_j) + C_R * card(SA(S_l)) < C_R * card(SA(S_r)) \quad (3)$$

The $card(SA(S_l))$ and $card(SA(S_r))$ values are found using the fragment access statistics histograms. We have used $C_R = 1$, i.e. a cost of 1 for accessing one tuple. Similarly, the cost of migrating a fragment F is equal to the number of tuples in the fragment, denoted $card(F)$. We therefore get:

$$card(F) + card(SA(S_l)) < card(SA(S_r)) \quad (4)$$

In order to compare different possible migrations, we use:

$$utility' = card(SA(S_r)) - card(SA(S_l)) - card(F) \quad (5)$$

While this equation is an expression of possible communication cost savings from a given migration, it can not quite be used as it is in the actual implementation. There are a couple of problems. First, SA by design includes only the recent history and the $card(SA)$ 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_F. This weight will have to be experimentally determined and optimal value will depend on how much history we allow SA to contain.

The second problem is that Eq. 5 could in some cases lead to an unstable situation where a fragment is migrated back and forth between sites. This is something we want to prevent as migrations causes delays in table accesses and indicies may have to be recreated every time. For a small fragment, especially with the introduction of w_F, just a few more accesses from S_r compared to S_l is enough to result in migration. If two sites alternate in having the upper hand with respects to number of accesses, unstable fragmentation would be the result. To alleviate this problem, we scale $card(SA(S_l))$ by w_{Sl}. A value $w_{Sl} = 1.5$ would mean that there have to be 50 % more remote accesses than local accesses for migration to be an option.

For small fragments with little traffic, w_{Sl} is not enough to achieve stability (e.g. $2 > w_{Sl} * 1$). To prevent this, we ensure that the number of remote accesses is above a percentage w_{SA} of the total number of accesses. The revised version of Eq. 5 becomes:

$$utility = card(SA(S_r) - w_{Sl} * card(SA(S_l)) \\ -w_{SA} * card(SA) - w_F * card(F) \quad (6)$$

Values for all these three cost function weights are experimentally determined in the Evaluation Section below.

7 Evaluation

In this section we present an evaluation where our approach is compared to two alternative fragmentation strategies. We compare by measuring the communication cost (remote accesses and migrations) for a series of distributed workloads. The evaluation has five parts. First we describe the experimental setup. Then we determine optimal values for the three cost function weights. Next, we examine how sensitive the evaluation result is to changes in the function weights. The optimal values for the weights are then used in an examination of ten simple workloads where each highlight a different aspect of how our approach works. Finally, we study in detail a more complex workload involving five sites.

7.1 Experimental setup

For the evaluation, we implemented a simulator which allowed us to generate distributed workloads, i.e., simulate several sites all performing tuple accesses with separate access patterns. For each simulated site, the following parameters could be adjusted:

- Fragmentation attribute value interval: minimum and maximum values in the accesses from the site.
- Access distribution: either uniform, hot spot (10 % of the values get 90 % of the accesses) or sequential access.
- Frequency: this could either be constant or with bursts. For bursts, 90 % of the operations were generated in 10 % of the time.
- Average rate of tuple accesses in number of accesses per minute. We use a Poisson distribution to generate accesses according to the frequency settings (constant or burst).

To our knowledge, no previous work exists that do dynamic refragmentation based on both reads and writes in a distributed setting. We have therefore chosen to compare our dynamic fragmentation approach to two alternative fragmentation methods:

The first is a baseline where *the table is not fragmented at all*. The table consists of a single fragment 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 (e.g. to simplify implementation and configuration), at least given that workloads were completely predictable. Since there is no fragmentation, 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 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 and frequency of accesses, neither of which are required for our dynamic approach.

7.2 Cost function weights

For initial testing, we designed 10 test cases. Each test case has two sites accessing 10000 tuples each. Early testing showed that 10000 tuples was more than enough to reach a stable fragmentation situation. Only two sites were used for these workloads in order to simplify workloads which would allow us to better detect how different workload parameters impact our results. Each workload was therefore designed with one specific purpose in mind. We later present a more complex workload involving five sites.

The fragmentation attribute value intervals for the two sites were designed so that they either overlap 100 %, 10 % or not at all. Two rates are used, a high rate of 1500 operations per minute and a low rate of 750 operations per minute. For the first seven test cases, the workload was constant for both sites, while the last three switched workload parameters halfway through the test case. These three serve as examples of dynamic workloads where access patterns are not constant and predictable. The results from these cases should illustrate if our approach's ability to adjust fragmentation at runtime result in communication cost savings.

The test cases are detailed in Table 3. Below is an explanation of what we wanted to test with each test case.

Test case 1: Since there are no overlap between the accesses from the two sites, we should end up with two fragments; one for each site. This is therefore a simple test to see if the algorithm is able to detect non-overlapping accesses.

Test case	Overlap	Distribution	Frequency	Rate	Purpose
1	0 %	Uniform	Constant	High	Detect non-overlap
2	100 %	Uniform	Constant	High	Baseline
3	100 %	Uniform	Burst	High	Effects of burst
4	100 %	Uniform	Constant	High / Low	Detect rate diff.
5	100 %	Uniform / Hot spot	Constant	High	Detect hot spots
6	100 %	Uniform / Sequential	Constant	High	Effects of sequential
7	10 %	Uniform	Constant	High	Detect overlap
8	100 %	Uniform	Constant	High	Detect overlap change
	10 %	Uniform	Constant	High	
9	100 %	Hot spot / Uniform	Constant	High	Detect distribution change
	100 %	Uniform / Hot spot	Constant	High	
10	100 %	Uniform	Constant	High / Low	Detect rate change
	100 %	Uniform	Constant	Low / High	

Table 3: Evaluation test cases.

Test case 2: With 100 % overlap, uniform distribution and constant rate, the two sites are completely equal. One site should therefore not be preferred over the other and the fragment should remain whole and not be migrated. This is therefore a test of stability.

Test case 3: With burst accesses, one site can temporarily dominate the other in terms of number of accesses. However since the distribution and overall rates are equal, there is little to gain from migration. This test case therefore serves as a more difficult test of stability than the previous case.

Test case 4: Here the two sites have different access rates. The fragment should therefore end up on the site with the highest rate.

Test case 5: In this case, 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.

Test case 6: Sequential access works against our approach since we assume that recent history will be repeated in the near future. We therefore expect this test case to perform worse than the above cases.

Test case 7: This case has only 10 % overlap between the accesses from the two sites. This overlap should be detected and placed on either site while the two non-overlapping sections should be migrated to their respective sites.

Test case 8: In this case the workload changes after 50 % of the accesses have been made. Essentially we change from test case 2 to 7 and the fragmentation should match this change.

Test case 9: Another case with workload change. This time two clients swap between having uniform access and hot spots.

Test case 10: The last case is to test how well the algorithm can detect a workload where the rate changes halfway through.

A wide variety of values for the three different cost function weights were tested using these test cases. In these tests the dynamic fragmentation algorithm

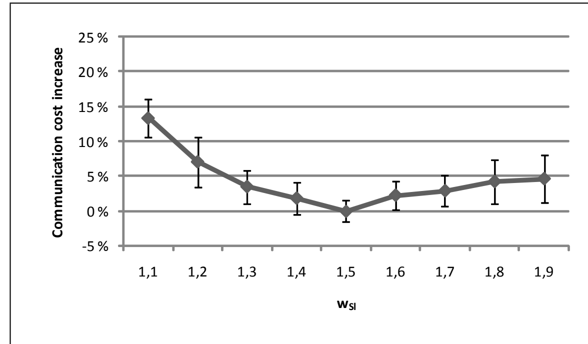

Figure 4: Communication cost as function of w_{Sl}.

was run every 30 seconds. From the results, it was determined that the best results were achieved with:

- $w_{Sl} = 1.5$
- $w_{SA} = 0.04$
- $w_F = 0.1$

7.3 Cost function weight sensitivity

Since the values for the three cost function weights were determined experimentally, it is important to examine to which degree small changes in weight values influence the result. Figures 4, 5 and 6 show how the communication cost increases with varying weight values compared to the best weight value set. Error bars indicate the standard deviation. Each point is an average from 60 runs of each of the 10 test cases described above.

The w_{Sl} weight controls the degree in which the number of local accesses should impact the chance of migration. With low values, many local accesses are needed to prevent migration and the migration cost thus dominated. With high values, only a few local accesses are needed to keep the fragment fixed and remote accesses will dominate the communication cost.

The second weight, w_{SA}, controls the percentage of accesses that must be from a given remote site before migration to this site is considered. With $w_{SA} = 0$, one more remote access than local access can be enough to force migration of a small fragment. So with low weight values, the situation is unstable with lots of migrations back and forth, leading to high communication cost. For high weight values, one remote site must dominate other sites with respect to access frequency

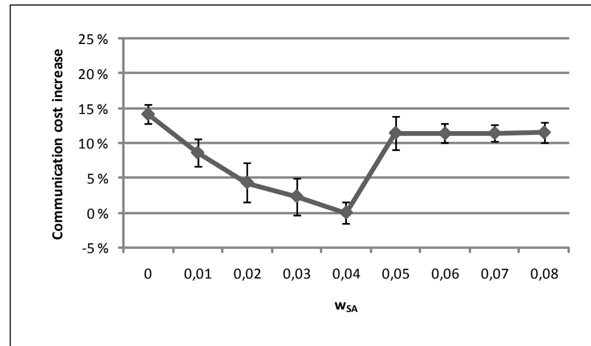

Figure 5: Communication cost as function of w_{SA}.

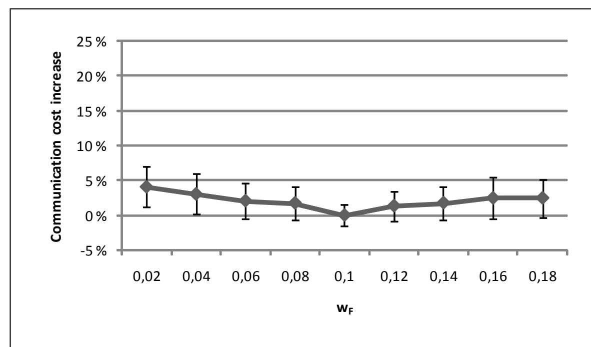

Figure 6: Communication cost as function of w_F.

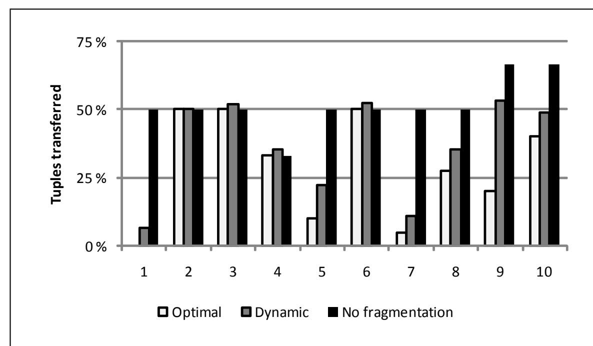

Figure 7: Detailed results, comparative view.

in order to force migration. This tends to lead to too few migrations and a large number of remote accesses.

w_F controls how important the fragment size is when migration is considered. For low weight values, large fragments will more easily be migrated and migration will dominate the communication cost. Similarly, large values will make large fragments all but immovable and remote accesses will be more prevalent.

7.4 Detailed results

Detailed results from the test with the best weight value set, is shown in Table 4. This table lists the number of remote accesses (out of 20000), migrations, fragments at the end of the run and the number of tuples transferred during migrations. The values are averages from 60 simulation runs. 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.

Test Case	Remote accesses	Migrations	Fragments	Tuples	Comm. cost	No fragmentation	Optimal
1	749	1	2	523	1272	10000	0
2	10000	0	1	0	10000	10000	10000
3	9982	20	4	431	10413	10000	10000
4	5123	5	2	199	5322	5000	5000
5	4038	11	20	422	4460	10000	2000
6	10132	4	2	354	10486	10000	10000
7	1674	1	2	494	2168	10000	1000
8	6199	1	2	896	7095	10000	5500
9	6821	51	21	1188	8009	10000	3000
10	6078	8	3	1258	7336	10000	6000

Table 4: Detailed results, best weight value set.

Fig. 7 show the results graphically. For each alternative algorithm, the graph shows the percentage of tuples transferred (either from remote writes or migrations).

Test case 1: Since we have 0 % overlap we should end with two fragments, one for each site. The results show that this is what happens. The only remote accesses that happen, are those that occur before the algorithm runs for the first time and the access pattern is detected.

Test case 2: With uniform distribution and 100 % overlap, it makes no sense to migrate anything as there is nothing to be saved wrt. locality. Note that the situation is stable, no migrations happen.

Test case 3: Burst causes some instability since it may appear from recent access history that one client is more active than the other. We therefore experience some migrations and fragmentation. This is an example of a workload not especially suited for our approach, even if the overhead compared to the other fragmentation strategies is minor.

Test case 4: With a rate difference, the fragment should always end up on the site with the highest rate. This happens, but it can take a little while for the algorithm to notice the pattern, especially if the first recorded accesses come from the low rate site.

Test case 5: Ideally, the algorithm should detect all hot spots and extract+migrate corresponding fragments while leaving the remainder of the table at the site with uniform distributed access. This is in fact what happens: 10 hot spots and 10 fragments.

Test case 6: A basic premise in our approach is that future accesses will be similar to recent accesses. With sequential access, this is not true. We therefore experience some minor overhead from wrongful migrations.

Test case 7: Similar to test case 1. The first time the algorithm runs, the 10 % overlap is noticed and the table fragmented accordingly. Only one migration so the situation is very stable.

Test case 8: Minor overhead from detecting the change in overlap. This overhead is larger than for test case 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.

Test case 9: Similar to test case 5 with extra overhead as for test case 8.

Test case 10: Similar to test case 4 with extra overhead as for test case 8.

7.5 Workload involving more sites

This section presents the results from a more comprehensive workload involving more than just two sites. The workload is illustrated in Fig. 8. It consists of five sites: two ”readers” and three ”writers”. Each of the writers accesses to a limited value interval of the fragmentation attribute, with uniform distribution and constant rate (3000 accesses per minute). They could, e.g., represent the writing of results

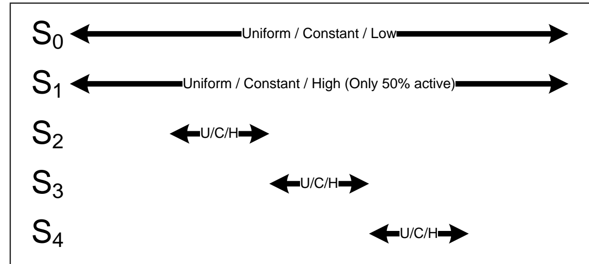

Figure 8: Workload with five sites.

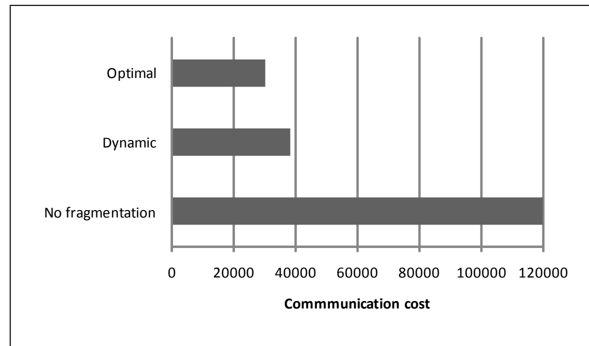

Figure 9: Comparative results from five site workload.

from a simulator distributed to three sites. The two readers access the whole table with uniform distribution. The first has a constant low rate of 1500 accesses per minute, while the second only starts after 50 % of the accesses have been made and then reads at a rate of 3000 accesses per minute.

The optimal fragmentation for this workload would be to fragment the table into five fragments: three for each of the sections accessed by the writers. The remaining two fragments should initially be allocated to S_0, but then be migrated to S_1 as soon as that reader starts (since it has higher rate than the reader on S_0). The results from the simulation of our dynamic fragmentation algorithm applied to this workload is shown in Fig. 9. For each simulation run, the sites accessed 160 000 tuples and the run was completed 60 times. While no fragmentation has 300 % higher network cost than optimal fragmentation, our algorithm was only 27 % higher than optimal. Some of this overhead is unavoidable as it takes a little while to detect access patterns (remember that the optimal algorithm assumes perfect knowledge about the future.)

For this workload we also tested how the size of the access history and the interval between fragment evaluations affect the results. Since $histogramReorganize()$

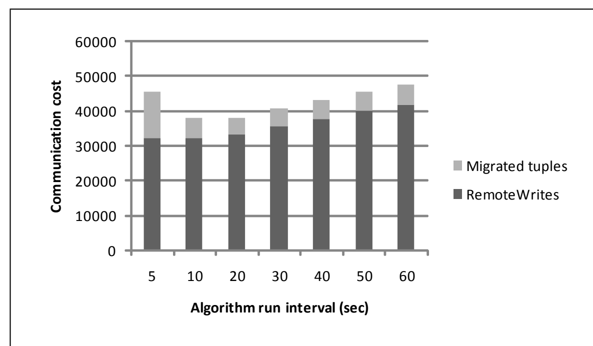

Figure 10: Effect of altering algorithm run interval.

(see Algorithm 5) is executed every time the fragmentation algorithm runs, the access data is used two times. Once as part of the current histogram and once in the old histogram. In other words, a fragmentation algorithm interval of 30 seconds means that 60 seconds of access history is evaluated each time. We tested algorithm intervals between 5 and 60 seconds and the results are shown in Fig. 10.

From these results, we can see that using a limited history means more migrations (instability). This makes sense as the access history then becomes more prone to random fluctuations in the access patterns. We also see that running the algorithm with longer intervals leads to more remote writes as new access patterns are not acted upon as quickly.

8 Conclusions and further work

In distributed database systems, tables are frequently fragmented over a number of sites in order to reduce access costs. How to fragment 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 query analysis. In this paper we have presented *DYTAF*, 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 our knowledge, no previous work exists that do dynamic refragmentation based on both reads and writes in a distributed setting.

Results from simulations show that for typical workloads, our dynamic fragmentation approach performs close to optimal fragmentation with respect to communication costs. Especially the ability to detect and adapt to workload changes show promise.

Future work include exploring adaptive adjustment of the cost function weights

as well as better prediction of workload based on control theoretical techniques. We also intend to use a variant of our approach to detect parts of fragments that have only read accesses, so that these parts can be extracted and replicated in order to improve global read performance.

References

[1] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of SIGMOD'2006*, 2006.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, 2004.

[3] I. Ahmad, K. Karlapalem, Y.-K. Kwok, and S.-K. So. Evolutionary algorithms for allocating data in distributed database systems. *Distrib. Parallel Databases*, 11(1), 2002.

[4] P. M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 13(3):263–304, 1988.

[5] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proceedings of ICDE*, 2007.

[6] A. Brunstrom, S. T. Leutenegger, and R. Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Proceedings of CIKM '95*, 1995.

[7] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of SIGMOD'1988*, 1988.

[8] A. L. Corcoran and J. Hale. A genetic algorithm for fragment allocation in a distributed database system. In *Proceedings of SAC'94*, 1994.

[9] D. Donjerkovic, Y. E. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proceedings of ICDE*, 2000.

[10] R. S. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *Proceedings of SIGMOD'1978*, 1978.

[11] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of DOLAP'2004*, 2004.

[12] B. Gavish and O. R. L. Sheng. Dynamic file migration in distributed computer systems. *Commun. ACM*, 33(2):177–189, 1990.

[13] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of VLDB'1990*, 1990.

[14] Y. Ioannidis. The history of histograms (abridged). In *Proceedings of VLDB'2003*, 2003.

[15] M. Ivanova, M. L. Kersten, and N. Nes. Adaptive segmentation for scientific databases. In *Proceedings of ICDE'2008*, 2008.

[16] S. Menon. Allocating fragments in distributed databases. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):577–585, 2005.

[17] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *Proceedings of SIGMOD'2002*, 2002.

[18] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1):29–56, 1985.

[19] D.-G. Shin and K. B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Trans. Software Eng.*, 17(9):872–883, 1991.

[20] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[21] G. Weikum, C. Hasse, A. Moenkeberg, and P. Zabback. The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5):381–432, 1994.

[22] E. Wong and R. H. Katz. Distributing a database for parallelism. *SIGMOD Rec.*, 13(4):23–29, 1983.

[23] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *Proceedings of VLDB*, 2004.