Technical Report RT/08/2014

AutoPlacer: Scalable Self-Tuning Data Placement in Distributed Key-value Stores

João Paiva
INESC-ID/IST
jgpaiva@gsd.inesc-id.pt

Pedro Ruivo
INESC-ID/IST
pruivo@gsd.inesc-id.pt

Paolo Romano
INESC-ID/IST
romano@inesc-id.pt

Luis Rodrigues
INESC-ID/IST
ler@inesc-id.pt

April 2014

Abstract

This paper addresses the problem of self-tuning the data placement in replicated key-value stores. The goal is to automatically optimize replica placement in a way that leverages locality patterns in data accesses, such that inter-node communication is minimized. To do this efficiently is extremely challenging, as one needs not only to find lightweight and scalable ways to identify the right assignment of data replicas to nodes, but also to preserve fast data lookup. The paper introduces new techniques that address these challenges. The first challenge is addressed by optimizing, in a decentralized way, the placement of the objects generating most remote operations for each node. The second challenge is addressed by combining the usage of consistent hashing with a novel data structure, which provides efficient probabilistic data placement. These techniques have been integrated in a popular open-source key-value store. The performance results show that the throughput of the optimized system can be six times better than a baseline system employing the widely used static placement based on consistent hashing.

AutoPlacer: Scalable Self-Tuning Data Placement in Distributed Key-value Stores

João Paiva Pedro Ruivo Paolo Romano Luís Rodrigues
INESC-ID Lisboa / Instituto Superior Técnico, Universidade Técnica de Lisboa
{jgpaiva,pruivo}@gsd.inesc-id.pt,{romano,ler}@inesc-id.pt

May 12, 2014

1 Introduction

Distributed NoSQL key-value stores [17, 27] have emerged as the reference architecture for data management in the cloud. A fundamental design choice in these distributed data platforms is how to determine the placement of objects (i.e., key/value pairs) among the nodes of the system. A data placement algorithm must simultaneously address two main, typically opposing, concerns: i) maximizing locality by replicating data at the nodes that access them more frequently, while enforcing constraints on the object replication degree and on the capacity of nodes; ii) maximizing lookup speed, by ensuring that a copy of an object can be located as quickly as possible.

The data placement problem has been investigated in a number of alternative variants, e.g. [20, 26]. Classic approaches formulate the data placement problem as a constraint optimization problem, and use Integer Linear Programming techniques to identify the optimal placement strategy with the granularity of single data items. Unfortunately, these approaches suffer from several practical limitations. In first place, finding the optimal placement is a NP-hard problem, hence any approach that attempts to optimize the placement of each and every item is inherently non-scalable. Further, even if the optimal placement could be computed, it is challenging to maintain efficiently a (potentially very large) directory to store the mapping between items and storage nodes.

Directories are indeed used by several systems such as PNUTS [12] or BigTable [8]. To minimize the costs associated with the maintenance of the directory, these systems trade-off placement flexibility and support placement at a very coarse level, i.e. large data partitions rather than on a per instance basis. However, even if coarse granularity is used, the use of a directory service introduces additional round-trip delays along the critical execution path of data access operations, which can hinder performance considerably.

To avoid the above issues, many popular key-value stores, such as Cassandra [27], Dynamo [17], Infinispan [34], use random placement based on consistent hashing. By relying on random hash functions to determine the location of data across nodes, these solutions allow lookups to be performed locally, in an very efficient manner [17]. However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), solutions based on consistent hashing may result in highly sub-optimal data placements.

This paper presents AutoPlacer, a system aimed at self-tuning the data placement in a distributed key value store, which introduces a set of novel techniques to address the trade-offs described in the previous paragraphs. Unlike conventional solutions [20, 26], that formulate the data placement optimization problem as an intractable ILP problem, AutoPlacer employs a lightweight self-stabilizing distributed optimization algorithm. The algorithm operates in rounds, and, in each round, it optimizes, in a decentralized fashion, the placement of the top-k "hotspots", i.e. the objects generating most remote operations, for each node of the system. This design choice has the advantage of reducing the number of decision variables for the data placement problem (solved at each round), ensuring its practical viability.

In order to be able to identify the "hotspots" of each node with low processing cost, AutoPlacer adopts a state of the art stream analysis algorithm [35] that permits to track the top-k most frequent items of a stream in an approximate, but very efficient manner. The information provided by the Space-Saving

Top-k algorithm is then used to instantiate the data placement optimization problem. We first study the accuracy of the solution from a theoretical perspective, deriving an upper bound on the approximation ratio with respect to a solution using exact frequencies. Next we discuss how to maximize the efficiency of the solution, showing how it can be made amenable for being partitioned in independent sub-problems, solvable in parallel.

Unlike solutions that rely on directory services, AutoPlacer guarantees 1-hop routing latency. To this end, AutoPlacer combines the usage of consistent hashing, which is used as the default placement strategy for less popular items, with a highly efficient, probabilistic mapping strategy that operates at the granularity of the single data item, achieving high flexibility in the relocation of (a possibly very large number of) hotspot items.

The key innovative solution introduced to pursue this goal is a novel data structure, which we named *Probabilistic Associative Array* (PAA). The goal of the PAA is to minimize the cost of maintaining a mapping associating keys with nodes in the system. PAAs expose the same interface of conventional associative arrays, but, in order to achieve space efficiency, they trade-off accuracy and rely on probabilistic techniques which can lead to inaccurate results with a user-tunable probability (these inaccuracies do not affect the correctness of the system, in worst case they may only degrade its performance). Internally, PAAs rely on Bloom Filters (BFs) and on Decision Tree (DT) classifiers. BFs are used to keep track of the elements inserted so far in the PAA in a space-efficient way; DTs are used to infer a compact set of rules establishing the associations between keys and values stored in the PAA. In order to maximize the effectiveness of the DT classifier, we expose a programmatic API that allows programmers to provide semantic information on the nature of the keys stored in the PAA (e.g., the data type of the value associated with the key). This information is then exploited, during the learning phase of the PAA's DT, to map keys into a multi-dimensional space that can be more effectively clustered by a DT classifier.

In summary, AutoPlacer provides two key features:

- It introduces a novel iterative, decentralized, self-tuning data placement optimization scheme.
- It preserves efficient lookups, while achieving high flexibility in determining an optimized data placement, through the use of a new probabilistic data structure designed specifically for this purpose.

AutoPlacer has been integrated in a popular, open-source key-value store, namely Infinispan: Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. The choice of Infinispan is also motivated by the fact that the authors are involved in a European project [41], which used Red Hat's Infinispan as the backbone of a highly scalable and self-tuning cloud data store. We conducted experiments on both public and private cloud infrastructures, using a porting of the well-known TPC-C benchmark for key-value stores, and GeoGraph [50], a complex benchmark representative of Geo-social network applications. The results of our experimental study highlight the effectiveness of AutoPlacer, which can achieve up to $6x$ speed-ups with respect to a baseline system using consistent hashing.

The remaining of the paper is structured as follows. Our target system is characterized in Section 2. Section 3 provides a global overview of AutoPlacer. Then, its components are described in more detail in the next two sections: the PAA internals are described in 4; a theoretical analysis of the optimizer's accuracy is provided in 5. Section 6 reports the results of the experimental evaluation of the system. Section 7 compares our system with related work. Finally, Section 8 concludes the paper.

2 System Characterization

The development of AutoPlacer has been motivated by our experience [39, 38, 42] with the use of an existing, state-of-the art, key-value store, namely Infinispan [34] by Red Hat©. In Infinispan (and other similar products such as [27, 17]), data is stored in multiple nodes using consistent hashing. For each key, consistent hashing determines a *supervisor* node for that item. Items can be replicated. A node that stores a copy of data item i is denoted an *owner* of that item. Assume that d copies are maintained of each data item, the owners of data item i are deterministically assigned to be j's supervisor plus its $d-1$ immediate successors (in the one hop distributed hash table that is used to implement consistent hashing).

Each node serves a dual purpose: it stores a subset of the data items maintained by the distributed store and also executes application code. The application code may be structured as a sequence of *transactions* (Infinispan supports transactional properties), with different isolation levels.

When the application code reads a data item, its value must be retrieved from one of its owners (which can be another node in the cluster). Thus, optimal performance is achieved if the node that executes a given application is the owner for the items it accesses more often. When the application writes a data item, all owners must be updated. Interestingly, the placement policy can also affect the performance of write operations. When multiple writes are performed in the context of a transaction, they can be applied in batch when the transaction commits. Hence, the larger the number of owners of keys updated by a transaction, the higher the number of nodes that have to be contacted during its commit phase.

Infinispan uses consistent hashing to ensure that all lookups can be executed locally. Unfortunately, in typical deployments of large-scale key-value stores, random data placement can be largely suboptimal as applications are likely to generate skewed access distributions [30], often dependent on the actual "type" of operations processed by each node [48, 16]. Also, workloads are frequently distributed according to load balancing strategies that strive to maximize locality [23]/minimize contention [2]. As we will show in the evaluation section, all these facts make consistent hashing sub-optimal. Therefore, significant performance improvements can be achieved by using appropriate autonomic data placement strategies.

3 AutoPlacer Overview

AutoPlacer is designed to optimize data location in a decentralized manner, i.e., each node in the system contributes to the global optimization process. Since AutoPlacer is aimed at systems that use consistent hashing as the default data placement policy, we also rely on consistent hashing to decentralize the optimization effort: each node is responsible for deciding the placement for the items it supervises. AutoPlacer executes, periodically, a sequence of optimization rounds. As a result of each round, a number of data items may be relocated. This happens only if the expected gains are above a minimum threshold. Each optimization round consists of the following sequence of six tasks.

Task 1: The first task of the AutoPlacer approach consists of collecting statistics about the *hotspots* data, i.e., the top-k most accessed data items, at each node. In fact, instead of trying to optimize the placement of every data item in a single round, at each optimization round, AutoPlacer only optimizes the placement of items that are identified as hotspots. Since this task is run periodically, once some hotspots have been identified (and relocated) in a given round, new (different) hotspots are sought in the next round. Therefore, although in each round only a limited number of hotspots is identified, in the long run, a large number of data items may be selected over multiple optimization rounds, as long as gains can still be obtained from their relocation.

Task 2: The second task consists in having the nodes exchange statistics regarding the data items that were identified as hotspots during the current round. More precisely, each node gathers (from the remaining nodes of the platform) access statistics on any hotspot items it supervises.

Task 3: The above information is used in the third task (denoted the *optimization* task) to find an appropriate placement for those items. The result of this task is a *partial relocation map*, i.e., a mapping of where replicas of each hotspot items that the node supervises (for the current round) must be placed.

Task 4: Even if the number of hotspots tracked at each round is a small fraction of the entire set of items maintained in the key-value store, over multiple rounds the relocation map can grow in an undesired way, and may even be too large to be efficiently distributed to all nodes. This task is devoted to encoding the relocation map in a probabilistic data structure that can be efficiently replicated on all nodes in order to ensure fast lookups, i.e. a *Probabilistic Associative Array* (PAA). Specifically, each node computes the PAA for the (relocated) objects it supervises.

Task 5: Once each PAA has been computed, each node disseminates it among all nodes. By assembling the PAAs received from all the nodes in the system, each node can locally build an object lookup table that includes updated information on the placement of data optimized during this round.

Task 6: Finally, at the end of each round, the data items for which new locations have been derived are transferred (using conventional state-transfer facilities [25, 43]) in order to match the new data placement.

As can be inferred from the previous description, the work is divided among all nodes and communication

Table 1: Parameters used in the ILP formulation.

$\mathcal{N}$	the set of nodes j in the system
$\mathcal{O}$	the set of objects i in the system
X	a binary matrix in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise
r_{ij}, w_{ij}	the number of read, resp. write, accesses performed on a object i by node j
cr^r, cr^w	the cost of a remote read, resp. write, access
cl^r, cl^w	the cost of a local read, resp. write, access
d	the replication degree, that is number of replicas of each object in the system
S_j	the capacity of node j.

takes place only during tasks 2, 5, and 6, in order to, respectively, exchange statistical information on hotspots, distribute the PAA, and finally relocate the objects. Also the tasks that require communication are performed in parallel, without the help of any centralized component.

Note that AutoPlacer is only concerned with the placement of data and is agnostic to the data consistency mechanisms used by the key-value store. For instance, AutoPlacer is compatible with the different consistency levels supported in Infinispan. Naturally, the consistency protocol in use must be prepared to support dynamic re-adjustment of the number and placement of replicas, but this is today the default for most consistency protocols, as it is required anyway for fault-tolerance.

In the next subsections we provide more information about the two main components of AutoPlacer, namely, the optimizer (executed by Task 3) and the PAA (built in Task 4 and used subsequently to perform data lookups locally).

3.1 Optimizer

Most works, e.g., [48, 29, 26, 20], in the area of data placement (and of its many variants [26, 20]) assume that the objective and constraint functions of the optimization problem can be expressed (or approximated) via linear functions, and accordingly formulate an Integer Linear Programming (ILP) problem. The ILP model can indeed be adopted also for the specific data placement problem tackled in this paper. To this end, one can model the assignment of data to nodes by means of a binary matrix X, in which $X_{ij} = 1$ if the object i is assigned to node j, and $X_{ij} = 0$ otherwise. Further, one can associate (average, or per object) costs with local/remote read/write operations. The ILP problem is then formulated as the minimization of the objective function that expresses the total cost of accessing all data items across all nodes, subject to two constraints: i) the number of replicas of each object must meet a predetermined replication degree, and ii) each node has a finite capacity (it must not be assigned more objects than it can store). In Table 1 we list the parameters used in the problem formulation, which aims at minimizing the following cost function:

$$\sum_{j\in\mathcal{N}}\sum_{i\in\mathcal{O}} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij}) \tag{1}$$

subject to:

$$\forall i \in \mathcal{O} : \sum_{j\in\mathcal{N}} X_{ij} = d \wedge \forall j \in \mathcal{N} : \sum_{i\in\mathcal{O}} X_{ij} \leq S_j$$

Despite its convenient mathematical formulation, ILP problems are NP-hard. Further, solving the above ILP problem would require to collect and exchange among nodes access statistics for all objects in the system. We tackle these drawbacks by introducing a lightweight, multi-round distributed optimization algorithm, which we describe in the following.

3.1.1 Space-Saving Top-k algorithm

An important building block of AutoPlacer is the Space-Saving Top-k algorithm by Metwally *et al.* [35]. This algorithm is designed to estimate the access frequencies of the top-k most popular objects in an approximate, but very efficient way, i.e. by avoiding maintaining information on the access frequencies (namely counters) for each object in the stream. Conversely, the Space-Saving Top-k algorithm algorithm maintains a tunable, constant, number m, where $m \ll |\mathcal{O}|$, of counters, which makes it extremely lightweight. On the downside, the information returned in the top-k list may be inaccurate in terms of both the elements that compose it and their estimated frequency. However, this algorithm has a number of interesting properties concerning the inaccuracies it introduces. First, it ensures that the access frequencies of the objects it tracks are always consistently overestimated. Also, its maximum overestimation error is known, and is equal to the frequency of the least frequently accessed item present in top-k, denoted as F_k. Finally, its space-requirements can be tuned to bound the maximum error introduced in the frequency tracking, as we will further discuss in Section 5.

3.1.2 Using Approximate Information

In AutoPlacer each node j runs 2 distinct instances, noted as $\textit{top-}k_j^{rd}$, resp. $\textit{top-}k_j^{wr}$, of the Space-Saving Top-k algorithm, used to track the k most frequently read, resp. updated, data items during the current optimization round. We denote with $\textit{top-}k_j(\mathcal{O})$ the subset of cardinality k (of the entire data set $\mathcal{O}$) contained in both the read and write top-k instances at node j, and with $\textit{top-}K(\mathcal{O}) = \cup_{j\in\mathcal{N}}(\textit{top-}k_j(\mathcal{O}))$ the union of the top-k data items across all nodes.

By restricting the optimization problem to the top-k accessed data items we reduce the number of decision variables of the ILP problem significantly, namely from $|\mathcal{O}||\mathcal{N}|$ to $O(k|\mathcal{N}|)$ (where $k \ll |\mathcal{O}|$). This choice is crucial to guarantee the scalability of the proposed approach. However, it requires to deal with the incomplete and approximate nature of the data (read/write) access statistics provided by the top-k algorithm, which we denote with $\hat{r}_{ik}$,$\hat{w}_{ik}$ to distinguish them from their exact counterparts (r_{ik},w_{ik}). Also, we use the notation $\hat{X}$ to refer to the solution of the optimization problem using as input the access statistics provided by the top-k algorithm, and distinguish it from the one obtained using the exact access statistics in input, which we denote X^{opt}.

A first problem to address is related to the possibility of lacking information concerning the access frequency by some node j for some data item $i \in \textit{top-}K(\mathcal{O})$: this can happen in case i has not been tracked in $\textit{top-}k_j(\mathcal{O})$, but is present in the $\textit{top-}k_{j'}(\mathcal{O})$ of some other node $j' \neq j$. To address this issue, we simply set to 0 the frequencies $\hat{r}_{ij}$,$\hat{w}_{ij}$

Finally, the approximate nature of the information provided by the Space-Saving Top-k algorithm may impact the quality of the identified solution. A theoretical analysis aimed at evaluating this aspect will be provided in Section 5.

3.1.3 Accelerating the solution of the optimization problem

To accelerate the solution of the optimization problem we take two complementary approaches: relaxing the ILP problem, and parallelizing its solution.

The ILP problem requires decision variables to be integers and is computationally onerous [48]. Therefore, we transform it into an efficiently solvable linear programming (LP) problem. To this end, we let the matrix $\hat{X}$ assume real values between 0 and 1 (adding an extra constraint $\forall i \in \mathcal{O}, \forall j \in \mathcal{N}\ 0 \leq \hat{X}_{ij} \leq 1$). Note that the solutions of the LP problem can have real values, hence each object is assigned to the d nodes which have highest $\hat{X}_{ij}$ values. As in [48], we use a greedy strategy according to which, if the assignment to a node causes a violation of its capacity constraint, the assignment is iteratively attempted to the node that has the $d + k$-th ($k \in [1, |\mathcal{N}| - d]$) highest scores.

Second, we introduce a controlled relaxation of the capacity constraint, which allows us to partition the ILP problem into $|\mathcal{N}|$ independent optimizations problems that we solve in parallel across the nodes of the platform. Let $\textit{top-}k_j(\mathcal{O}|n)$ be the set of keys in $\textit{top-}k_j(\mathcal{O})$ of node j that node n supervises. Each node j sends its $\textit{top-}k_j(\mathcal{O}|n)$ to each other node n in the system. As a result each node j also gathers the access statistics $\textit{top-}K(\mathcal{O}|j) = \cup_{n\in\mathcal{N}}\ \textit{top-}k_n(\mathcal{O}|j)$ concerning the current hotspots that j supervises. At this point each node j computes the new placement for the data in $\textit{top-}K(\mathcal{O}|j)$.

Table 2: PAA Interface.

Method	Input Parameters	Output
CREATE	Set⟨Key,Value[d]⟩, α, β	PAA
GET	Key	Value[d]
ADD	Set⟨Key,Value[d]⟩	PAA
GETDELTA	PAA	ΔPAA
APPLYDELTA	ΔPAA	PAA

Note that since we are instantiating the (I)LP optimization problems in parallel, and in an independent fashion, we need to take an additional measure to guarantee that the capacity constraints are not violated. To this end we instantiate the (I)LP problems at each node j with a capacity $S'_j = S_j - |\mathcal{N}|k$. In practice, this relaxation is expected to have minimum impact on the solution quality as $k \ll S_j$.

Overall, at the end of an optimization round each node j produces two outputs: the partial relocation map $\hat{X}$, and the cost reduction achievable by relocating the data in *top-K*$(\mathcal{O}|j)$ according to $\hat{X}$, which we denote as Δ_{C^j}. Δ_{C^j} is computed as the difference between the result of Equation 1 applied to the partial relocation map $\hat{X}$ obtained in this round, and that obtained in the previous round. This value allows estimating the gain achievable by performing this optimization round, and, as we will discuss shortly, is used in AUTOPLACER to determine the completion of the round-based optimization algorithm.

3.2 Probabilistic Associative Array: Abstract Data Type Specification

Even though in each round AUTOPLACER optimizes the placement of a relatively small number of data items, over multiple optimization rounds the number of relocated objects can grow very large. Hence, a relevant issue is related to the overhead for maintaining, and replicating, a possibly very large relocation map. Indeed the relocation map can be seen as an associative array in which each entry is a pair mapping a data item to the set of nodes that own it.

The Probabilistic Associative Array (PAA) is a novel data structure that allows maintaining an associative array in a space efficient, but approximate way. We present the PAA as an abstract data type, with an interface analogous to conventional associative arrays. Later in Section 4, we will discuss how it has been implemented in AUTOPLACER.

The PAA is characterized by the API reported in Table 2, which is similar to that of a conventional associative array, including methods to create and query a map between keys and (constant d-sized) arrays of values. To this end, the PAA API includes three main methods:

- the CREATE method, which returns a new PAA instance and takes as input a set of pairs in the domain (key $\times$ array[d] of values) to be stored in the PAA (called, succinctly, *seed map*) and two tunable error parameters α and β (discussed below);
- the GET method, which allows querying the PAA obtaining the array of values associated with the key provided as input parameter, or $\perp$ if the key is not contained in the PAA;
- the ADD method, which takes an input a seed map and adds it to an existing PAA.

The PAA is designed to trade off accuracy for space efficiency, and may return inaccurate results when queried. In the following we specify the properties ensured by the GET method of a PAA:

- it may provide *false positives*, i.e., it can provide a return value different from $\perp$ for a key that was not inserted in the PAA. The probability of false positives occurring is controlled by parameter α.
- it has no *false negatives*, i.e., it will never return $\perp$ for a key contained in the seed map.
- it may return an *inaccurate* array of values for a key contained in the seed map. The probability of returning inaccurate arrays is controlled by parameter β. In other words, with some small and tunable probability, the data items may be located in different nodes than those specified by the seed map (thus, the efficiency of lookup may cause some degree of sub-optimal placement).

- its response is deterministic, i.e., for a given instance of a PAA, the return value for any given key is always the same.

Finally, the PAA API contains two additional methods that allow to update the content of a PAA in an incremental fashion: getDelta, and applyDelta. getDelta takes as input a PAA and returns an encoding, denoted as ΔPAA, of the differences between the base PAA over which the method is invoked (say PAA_1) with respect to the input PAA, say PAA_2. The ΔPAA returned by getDelta can then be used to obtain PAA_2 by invoking the method applyDelta over PAA_1 and passing as input parameter ΔPAA.

Before presenting the internals of the PAA implementation (see Section 4), we discuss, in the next sections, the functioning of AutoPlacer's distributed algorithm.

3.3 The AutoPlacer iterative algorithm

We now provide, in Algorithm 1, the pseudo-code formalizing the behavior of the AutoPlacer algorithm executed by a node j. Each node maintains a local lookup table, denoted as *LookupT*, that consists of an array of PAAs, one per each node j in the set of nodes composing the system (denoted by Π in Alg. 1). Specifically, j's entry of *LookupT* is used to keep track of the objects supervised by node j that have already been relocated by AutoPlacer. For any given round, *LookupT* is the same on all nodes.

At the beginning of each round, j collects statistics concerning its top-k most frequently read/written data items. This activity is encapsulated by the `collectStats` procedure (line 5), which is designed to track only accesses to objects whose placement had not been previously optimized in previous rounds. This measure is necessary, as, otherwise, in presence of stable distributions of the data access among nodes (i.e., stable workloads), the top-k lists at each node may quickly stagnate. Especially in case of skewed distributions the top-k lists would tend to track the very same objects (i.e., the most popular ones) along every round.

By tracking only the keys whose placement has not been optimized in previous rounds, it is guaranteed that, in two different rounds, two disjoint sets of objects are considered by the optimization algorithm, leading to the analysis of progressively less "hot" data items. Further, it prevents the possibility of ping-pong phenomena [21], i.e. the continuous re-location of a key between nodes, as it guarantees that the position of each object is optimized at most once.

To determine whether an access to a data item should be traced or not, the `collectStats` procedure is provided with *LookupT* as input (we recall that *LookupT* keeps track of all items whose placement has been previously optimized). Upon a read/write access on a data item, the `collectStats` procedure checks if the item is contained in *LookupT* and, in the positive case, it avoids tracing this access. Notice that this assumes that the data access frequencies do not change significantly during the entire optimization process. In fact, in order to cope with scenarios in which applications' data access patterns change at a frequency higher than AutoPlacer's complete optimization, the structure of *LookupT* must be more complex. In Section 3.4 we detail how these scenarios are handled by AutoPlacer.

Next the nodes exchange the information collected by `collectStats`. Since we also parallelize the optimization procedure, we send to each node only the statistical information that will be relevant to the computation that will be performed at that node, i.e., the statistical information regarding the data items it supervises.

At this point, each node optimizes the placement for the objects it supervises (primitive `Optimize`), determining their new owners (encoded in the partial relocation map, denoted $\hat{X}$). The node also computes the reduction of the local cost function (denoted as Δ_{C^j}) that the new assignment brings.

Then, node j computes a temporary PAA, based on the previous value of its PAA (stored in *LookupT*$[j]$) and on the new additional relocation information $\hat{X}$ (lines 13-14). The API of the PAA is then used to extract the relevant deltas from the existing PAA that need to be disseminated, in order to avoid sending the entire PAA again (line 15). These deltas are exchanged among nodes, and applied locally, such that every node can update all entries of *LookupT* (lines 16-22).

Each optimization round ends by triggering the re-location of the data via the `moveData`() primitive. This primitive will use the updated PAAs to determine the set of items that have been re-located, and gives the necessary commands to perform the corresponding state transfers. Several state transfer techniques could be used for this purpose [25, 43], whose complexity is dependant on the consistency guarantees that the key-value store implements (e.g. transactional vs eventual consistency). These mechanisms are indeed orthogonal to the AutoPlacer system.

ALGORITHM 1: AutoPlacer's behavior at node j

```
1  Array[1...|$\mathcal{N}$|] of PAA : LookupT={$\perp, \ldots, \perp$};
2  PAA: tmpPAA=$\perp$;
3  do
4      Array[1...|$\mathcal{N}$|] of Set$\langle i, r, w\rangle$ : req=$\perp$;
5      $\langle top_k^{rd}, top_k^{wr}\rangle \leftarrow$ collectStats(LookupT);
6      foreach $n \in \Pi$ do
7          send({$\langle i, r, w\rangle \in \{top_k^{rd} \cup top_k^{wr}\}$ such that supervisor($i$) = $n$}) to $n$;
8      end
9      foreach $n \in \Pi$ do
10         req[j]$\leftarrow$ receive() from $n$;
11     end
12     $\langle \hat{X}, \Delta_{C^j}\rangle \leftarrow$ Optimize(req);
13     tmpPAA $\leftarrow$ LookupT[$j$];
14     tmpPAA.ADD($\hat{X}$);
15     $\Delta$PAA: delta $\leftarrow$ tmpPAA.GETDELTA(LookupT[$j$]);
16     broadcast(delta,$\Delta_{C^j}$);
17     $\Delta_{C^*} \leftarrow 0$;
18     foreach $n \in \Pi$ do
19         [delta,$\Delta_{C^n}$] $\leftarrow$ receive() from $n$;
20         LookupT[$n$]$\leftarrow$LookupT[$n$].APPLYDELTA(delta);
21         $\Delta_{C^*} \leftarrow \Delta_{C^*} + \Delta_{C^n}$;
22     end
23     moveData();
24 while $\Delta_{C^*} > \gamma$;
```

AutoPlacer relies on a simple self-stabilizing mechanism that halts the distributed optimization algorithm if the "gain" achieved during the last optimization round does not exceed a user-tunable minimum threshold, denoted γ (line 23). This allows avoiding to analyze the "tail" of the data access distribution, whose optimization would lead to negligible gains. We chose as metric to evaluate the optimization gain the reduction of the cost function achieved during the last optimization round, Δ_{C^*}. To compute this value, each node j disseminates the value for the reduction of its local cost function Δ_{C^j} along with delta, in line 16. At the end of this dissemination phase each node of the system can deterministically compute Δ_{C^*} and evaluate the predicate on the termination of the optimization algorithm.

Finally, AutoPlacer leverages the fault-tolerant mechanisms used by the underlying key-value store itself to recover from node failures. In fact, Infinispan relies on JGroups [3] to maintain the cluster membership; if a node crashes both the key-value store and the AutoPlacer components receive consistent up-to-date membership change notifications that can be used to safely recover from failures (for instance, this prevents AutoPlacer from blocking while waiting from inputs from different nodes).

After one round of the AutoPlacer algorithm, some items will be mapped to nodes different from those mapped by consistent hashing. Hence, the system must use a lookup function which supports this change. Algorithm 2 shows the pseudocode for the lookup function for a key k. First, consistent hashing is used to identify the supervisor of k, say s. We then check whether the PAA associated with s contains k. In the positive case, we use the mapping information provided by $LookupT[s]$ to identify the set of nodes that are currently maintaining key k. Otherwise, we simply return the set of owners for k as determined by consistent hashing (d is the replication degree).

3.4 Handling dynamic workloads

In the previous sections we have described how AutoPlacer can be employed to optimize data placement in presence of static workloads. However, the algorithm can be extended to cope with dynamic workloads,

ALGORITHM 2: PAA-based lookup function

```
1 Array[1...d] of Nodes LOOKUP(Key k)
2     if LookupT[supervisor(k)].GET(k) ≠ ⊥ then
3         return LookupT[supervisor(k)].GET(k);
4     else
5         s ← supervisor(k);
6         return {s, s+1, ..., s+d-1};
7     end
8 end
```

i.e. scenarios in which the data access patterns generated by the nodes in the system vary over time. In this case, the data placement identified by AutoPlacer at the end of a round may become suboptimal and require the re-location of data items whose placement had previously been optimized by AutoPlacer. The key issue is that, once that a data item has been relocated by AutoPlacer, its data accesses are no longer traced via the top-k, in order to avoid re-optimizing its placement in following rounds and to prevent the stagnation of the top-k statistics. In the following we show how it is possible to extend AutoPlacer to cope with dyamic workloads, by having it operate in *epochs*.

In each epoch, AutoPlacer operates exactly as described in the previous sections. A new epoch is started when the need for recomputing data placement is identified, i.e. whenever an abrupt change of the access patterns is detected during the current epoch. Various key performance indicators may be adopted to reveal the occurrence of a relevant workload shift, including the ratio of remote to local data access operations and/or the number of nodes involved in commit. Indeed, as a consequence of AutoPlacer's optimization algorithm, these performance indicators are expected to decrease over time if the data access patterns at the various nodes remain stable.

In the prototype of AutoPlacer we trigger the beginning of a new epoch whenever we detect a sudden growth of the probability of remote data accesses. Each node disseminates this information periodically in the system. At each dissemination round, every node computes the average remote access probability in the system, and uses the same deterministic function (i.e., comparison with a user-configurable threshold) to determine whether to trigger a new epoch. Thus, the need to start a new epoch can be locally detected by any node. In such case, the node will simple send a message to the remaining nodes to make sure that every node will also re-start executing Alg. 1. This simple scheme ensures that all nodes will agree on triggering a new epoch in the same dissemination round.

Upon the beginning of a new epoch e, each node creates an empty $LookupT^e$ associated with the current epoch. However, unlike a normal AutoPlacer round, this $LookupT^e$ is not merged with any pre-existing $LookupT$s, but it is stacked on top of $LookupT$ from previous rounds. This new $LookupT^e$ will serve a twofold purpose. On one hand, it allows for storing a "patch" for the currently established data placement (encoded by the set of $LookupT$s of the preceding epochs). Further, the current $LookupT^e$ is provided as input to the `collectStats()` method, which will use it to determine whether or not to track accesses to a data item d in the node's top-k in the current epoch, depending on whether d is stored in $LookupT^e$ (which, we recall, implies that d has been re-located by AutoPlacer during epoch e).

Since in the first round of any epoch e, its $LookupT^e$ is empty, the accesses to *all* objects in the key-value store will be tracked, allowing to re-optimize the placement of data items that have been subjected to access patterns shifts since the last epoch. If objects need to be relocated, their new position is stored in the corresponding PAA in $LookupT^e$. At the end of the round, each node will broadcast the PAA associated with the data it supervises, as usual, and the other nodes will merge it with the empty $LookupT^e$. We alter also the way in which the lookup function operates: since an object may have been relocated in a previous epoch, we query the stack of $LookupT$ (in reverse chronological order). The lookup function returns the mapping determined by the first $LookupT$ that contains the required data item (i.e., the most recent re-location of a data item), or the result of the default consistent hashing scheme in case the data item has not been relocated in any epoch (and, therefore, is not present in any $LookupT$).

A simpler way for AutoPlacer to handle varying workloads would be to reset its $LookupT$ as it detects a change in the workload. Even though this would be an effective method of resetting AutoPlacer, it would have the negative effect of causing the system to shuffle all data previously optimized, returning to the original consistent hashing data placement. Keeping an history of previous *LookupT*, arranged in reverse chronological order and regulated by the notion of epochs, allows for minimizing the overhead introduced by the need to re-optimize the placement of data in the system. In fact, if the placement determined in a previous epoch remains optimal in the new epoch for a subset O of the platform's data, and is sub-optimal for a subset S, the data items in O will not be re-located in the new epoch (as the optimization algorithm executing during the new epoch will confirm the optimality of the current placement).

The disadvantage of keeping an history of *LookupT* is that it could slow down the lookup function after a long series of epochs. However, it is important to note that after the system has stabilized on the most recent epoch e and AutoPlacer has stopped optimizing placement, $LookupT^e$ will contain the entire set of hotspots given the data access patterns currently exhibited by the application. Thus, PAAs belonging to previous epochs will only be queried to obtain the location of objects which were relocated in some previous epoch, and that are no longer used by the nodes they were moved to. Hence, after AutoPlacer has stabilized, all but the most recent *LookupT* may be discarded, causing the non-hotspot objects to return to their supervisor nodes (as determined by consistent hashing). Since the AutoPlacer algorithm is executed independently at each node, each node n may unilaterally decide when to purge the previous PAAs associated with the data it supervises. So, when a node has finished optimizing placement, it broadcasts a `PURGE` message so that other nodes will discard the existing history of PAAs associated with the objects n supervises.

Finally, we note that in order to ensure the termination of the AutoPlacer's optimization process, we need to assume that the frequency with which workloads change is sufficiently low to give AutoPlacer enough time to execute a sufficient number of optimization rounds. However, even in challenging scenarios characterized by frequent shifts of the application's data access patterns, the algorithm described above allows for effectively reacting to workload changes by minimizing the costs associated with the re-mapping of data replicas to nodes.

4 Probabilistic Associative Array Internals

This section is devoted to discuss the implementation of PAAs. We start, in Section 4.1, by providing some background on two building blocks that are used internally to the PAA, namely Scalable Bloom filters [1], and a decision tree algorithm for data streams, called VFDT [19]. Next, in Section 4.2, we introduce the FeatureExtractor interface, a programming abstraction that allows to extract semantic information on the keys inserted in the PAA, and which aims to maximize the efficiency of its probabilistic, space-efficient encoding. In Section 4.3, we discuss in detail how each of the methods of the PAA's interface defined in Section 4.3 can be implemented. Finally, in Section 4.4, we illustrate with a concrete example how PAAs are exploited in the AutoPlacer system.

4.1 Building Blocks

Scalable Bloom filters (SBF) [1] are a variant of Bloom filters (BF) [6], a well know data structure that supports probabilistic test for membership of elements in sets. A BF never yields false negatives (if the query returns that an element was not inserted in a BF, this is guaranteed to be true). However, a BF may yield false positives (a query may return true for an element that was never inserted) with some tunable probability α, which is a function of the number of bits used to encode the BF and of the number of elements that one stores in it (that must be known a-priori). SBFs extend BFs in that they can adapt their size dynamically to the number of elements effectively stored, while still ensuring a bounded false positive probability. Internally, this is achieved by creating, on demand, a sequence of BFs with increasing capacity.

VFDT [19] is a classifier algorithm that induces decision trees over a stream of data, i.e. without assuming the a-priori availability of the entire training data set unlike most existing decision trees [36]. VFDT is an incremental online algorithm, given that it has a model available at any time during its run and refines the model over time (by performing new splits, or pruning unpromising leaves) as it is presented with additional training data. As classical off-line decision trees, the output of VFDT is a set of rules that allows to map a point in the feature space to a target discrete class. A noteworthy property of VFDT is that the trees it produces are asymptotically arbitrarily close to the ones produced by a batch learner (i.e., an off-line learner that uses the entire training set to determine how to grow the tree). The misclassification probability can be configured by means of the parameter δ [19], which affects the frequency with which new rules are induced in the trees built by the VFDT algorithm.

The PAA uses SBFs and VFDT in the following manner. SBFs are used to assert if a key was stored in the PAA. VFDT is used to obtain the set of values associated with a key stored in tha PAA. The next sections explain how this technique works in detail.

4.2 FeatureExtractor Key Interface

In order to maximize the effectiveness of the machine-learning statistical inference, programmers can optionally provide semantic information on the type of key inserted in a PAA, by having their keys implementing the FeatureExtractor interface. This interface exposes a single method, getFeatures(), which returns a set of pairs $\langle$*featureName*, *featureValue*$\rangle$, where *featureName* is a unique string identifying each feature and *featureValue* is a (continuous or discrete) value defining the value of that feature for the key.

The purpose of this interface is to allow a key to be mapped, in a semantically meaningful (and hence inherently application dependant) way, into a multi-dimensional feature-space that can be more efficiently analyzed and partitioned by a statistical inference tool. In particular, the set of features extracted via this interface defines the "feature space" [36] over which, as we will see in the following section, the VFDT algorithm infers a set of compact rules. This ruleset, which is encoded as a decision tree, allows defining which regions of the feature space should be assigned to the various nodes of the system.

Normally, features can be "naturally" derived from the data model used in the application. For instance, if an object-oriented (or relational) model is used, a typical encoding for the key corresponding to an object of class "Person" with ID=3 may be "`Person-3`". The FeatureExtractor interface can then simply parse the key and return the pair $\langle$"Person"$, id\rangle$. This can be further illustrated considering the real example of the TPC-C benchmark, which we used in our evaluation. In this case, a "Customer" object with id c_1 would be associated with a feature, $\langle$"Customer", $c_1\rangle$. Further, since in TPC-C a customer is statically registered in a "Warehouse" object, c_1 would have a second feature $\langle$"Warehouse", $w_1\rangle$, being w_1 the identifier of the warehouse where c_1 is registered. Hence, a different customer c_2 registered with a warehouse w_2 would be associated with the features $\langle$"Customer", $c_2\rangle$ and $\langle$"Warehouse", $w_2\rangle$, while the object representing warehouse w_2 itself would be associated with the features $\langle$"Customer", N/A$\rangle$ and $\langle$"Warehouse", $w_2\rangle$.

Note that this sort of feature extraction could be automated, provided the availability of information on the mapping between the application's domain model, in terms, e.g., of entities and relationships, and the underlying key/value representation. This can be done using annotations or domain specific languages, analogously to what is done in Object-to-Relational Mapping (ORM) solutions [4].

Finally, we note that this approach could be used, at least in principle, even in case of completely unstructured keys. In such a case, the feature extraction phase would map the keys onto a mono-dimensional feature space that coincides with the keys' domain. However, as we mentioned in Section 4.1, the PAAs use VFDT, an online decision-tree based classifier algorithm, to infer a set of rules that map regions of the feature space onto nodes in the system. As in any machine learning problem [5], the ability of the classifier to infer a compact, yet accurate rule set is, in practice, strongly dependant on the quality of the used features, and on their capability to promote the identification of clusters. Although, generally speaking, there is no universal rule that determines how many features should be used to achieve optimal performance [33], one may argue that, in most classification problems, using a mono-dimensional feature space is likely to limit the inference capabilities of classifier algorithms. This is the reason why AutoPlacer's programming model promotes, via the FeatureExtractor interface, the encapsulation of additional semantic information in the key structure.

4.3 PAA Operations

We can finally discuss how each of the methods specified by the PAA abstract interface introduced in Section 3.2 is implemented:

• Create: to instantiate a new PAA from a seed map, an SBF is first created, sizing it to ensure the target error rate α and populating it with the keys passed as input parameter. Next, d new instances of VFDT are trained. The i-th instance of VFDT ($i \in [1, \ldots, d]$) is trained by using a dataset containing, for each key k in the seed map, an entry composed by the mapping of k in the feature space (obtained using k's FeatureExtractor interface), and as target class value, the i-th value associated with k in the seed map. As we are creating a decision-tree from scratch over a fully-known training set, we use in this phase VFDT as an offline-learner, by configuring it to use the C4.5 algorithm [40]. This allows us to tightly control the cardinality of the rule set it generates to achieve arbitrary accuracy in encoding the mapping, and hence fine tune the pruning of the rule set to achieve the user specified parameter β. To this end we need to keep into account that, when new data is added to the PAA (via the Add method, to be discussed shortly), the decision tree is grown using the VFDT on-line algorithm, which can introduce an additional misclassification probability δ. Hence, during the pruning phase of the initial off-line trained decision tree, we aim at achieving a misclassification rate on the training set equal to $\beta' = 1 - \frac{1-\beta}{1-\delta}$.

The asymptotic complexity of the Create operation results as the sum of: i) creating an SBF — an operation that has constant cost, as it boils down to allocating a byte array whose size can be computed in O(1) based on α; ii) building the training set for d VFDTs via the FeatureExtractor interface — an operation that has $O(S \cdot f \cdot d)$ cost, where S is the number of keys in the seed map, f is the number of features encoded in the keys, and d is the replication degree; and iii) training d VFDTs, which has complexity $O(d \cdot (S \cdot f \cdot log(S))$, which is the cost of building d decision trees over a training set of cardinality S with f attributes using C4.5 [47].

• Get: queries for a key k are performed by first querying the SBF. If the response is negative, $\perp$ is returned. Otherwise (and this may be a false positive with probability α), the VFDT is queried by transforming k in its representation in the feature space by means of the FeatureExtractor interface. If k had actually been inserted in the PAA, the query to the SBF is guaranteed to return a correct result. However, it may still be wrongly classified by the VFDT algorithm, which may return any of the target classes that it observed during the training phase.

The asymptotic complexity of the Get operation is equal to $O(h \cdot sl + dt - depth)$ where h and sl are, respectively, the number of hash functions and of internal bloom filters used by the SBF, and $dt - depth$ is the depth of the decision tree induced by VFDT, which is typically assumed $O(log(S))$ when analyzing complexity of decision tree algorithms [47].

• Add: to implement this method, we leverage on the incremental features of the SBF and VFDT. To this end, we first insert each of the entries k passed as input parameter into the SBF. This may lead to the generation of an additional, internal bloom filter, to ensure that the bound on α is ensured. Next we incrementally train the VFDT instances currently maintained in the PAA, by providing them, in a single batch, the entire set of key/value pairs that is being added to the PAA. As the VFDT algorithm introduces at most δ misclassification errors with respect to an off-line decision tree, and given that we prune the initial, off-line built decision tree to achieve accuracy β', we guarantee the tree built by VFDT upon the execution of the Add method still achieves the target bound on misclassification β.

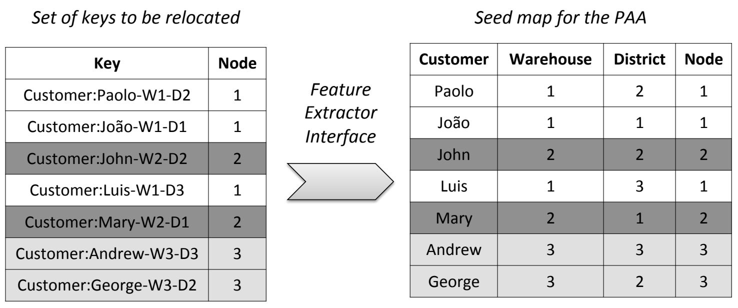
The asymptotic complexity of the Add operation is equal to $O(Sh + dSf\ log(S))$, which corresponds to the cost of inserting S items in the SBF (by using h hash functions), plus the cost of build d trees using VFDT over a batch of S examples in the training set.

• getDelta: the output consists of the binary diff of the SBFs, plus the rule set of the VFDT maintained by the PAA over which this method is invoked. To obtain the binary diff of the SBFs, we use a simple optimization that allows for avoiding the processing of all but the latest internal BF in common between the 2 PAAs being diffed. As new elements are added always to the most recent internal BF, when this method is invoked over a PAA p, passing as input parameter a PAA p', it returns: i) any new BF included in p as a result of the insertion of additional elements in p', and ii) the most recent BF stored by p', which may have been in the meanwhile altered to store additional elements.

Excluding, for simplicity, this optimization from the analysis, the complexity of the getDelta operation can be estimated as proportional to the size (measured in bytes) of the PAA over which this operation is invoked.

• applyDelta: symmetrically to what is done in getDelta, this method generates a new PAA, whose SBF is obtained by applying the binary SBF diff contained in the input ΔPAA to the SBF of the PAA over which this method is invoked. The rule set of the output PAA is set equal to the one contained in the input ΔPAA.

The complexity of this method, analogously to getDelta, is proportional to the size of the PAA passed


Set of keys to be relocated

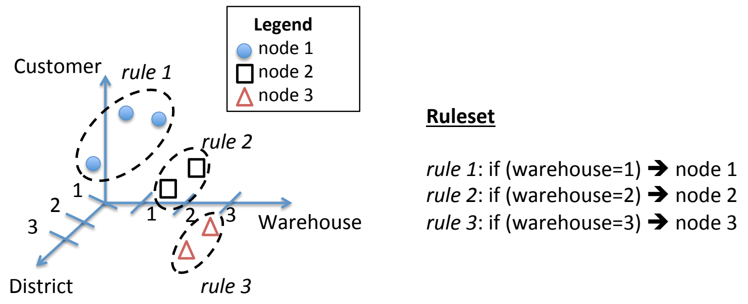
Key	Node
Customer:Paolo-W1-D2	1
Customer:João-W1-D1	1
Customer:John-W2-D2	2
Customer:Luis-W1-D3	1
Customer:Mary-W2-D1	2
Customer:Andrew-W3-D3	3
Customer:George-W3-D2	3

Feature Extractor Interface

Seed map for the PAA

Customer	Warehouse	District	Node
Paolo	1	2	1
João	1	1	1
John	2	2	2
Luis	1	3	1
Mary	2	1	2
Andrew	3	3	3
George	3	2	3

(a) Example usage of the FEATUREEXTRACTOR Interface.



Ruleset

rule 1: if (warehouse=1) ➔ node 1
rule 2: if (warehouse=2) ➔ node 2
rule 3: if (warehouse=3) ➔ node 3

(b) Mapping of the seed map in the feature space and rule-set inference via VFDT.

Figure 1: Example usage of PAAs in AUTOPLACER

as input parameter.

4.4 Example usage of PAAs in AutoPlacer

Figure 1 provides a concrete example of usage of PAA in AUTOPLACER. Let us start by analyzing Figure 1(a). On the left side, we report an example set of keys whose placement in the system needs to be updated, together with their corresponding new placement. We recall that this information is obtained as output of the optimization phase of each AUTOPLACER's round. Before inserting these keys in the PAA, the FEATUREEXTRACTOR interface is first executed, yielding the result on the right. This phase allows extracting semantic information embedded in the key structure, and to map them into a feature space, which, in the considered example, consists of tuples of the form $\{\text{CUSTOMER}\times\text{WAREHOUSE}\times\text{DISTRICT}\}$, and having as target class the identifier of the node to which the key should be reassigned.

Figure 1(b) illustrates the mapping of the seed map in the feature space. Note that in this diagram, we indicate with different point types (namely, a circle, a box and a triangle) keys that are mapped to different nodes/values of the target class. Note also that we omit to label the CUSTOMER axis for the sake of readability. The figure also depicts a possible clustering induced by the ruleset determined by running the VDFT algorithm on the seed map. In the considered example, the decision tree is built using three rules that partition the feature space according to the value of the WAREHOUSE feature, and attribute a different target class value (i.e., node) to each partition. In the considered example, which is clearly simplistic for the sake of clarity, the mapping of 7 keys to 3 different nodes (out of a possibly much larger set of total nodes) is encoded using exclusively 3 rules, which can be encoded in a much more compact way than the relocation map. It should be noted that the considered rule set may actually match a much larger number of keys than those included in the seed map. This would happen, for instance, in case each warehouse, in the considered example, was associated with a large population of customers.

This is the reason why PAAs integrate a SBF, which they use to store the ids of the keys, whose mapping has been inserted in the PAA. Assume that a PAA, following its initialization with the considered set of keys, is queried about a key k not included in this set, where $k = Customer : Pedro - W1 - D2$. With high probability — more precisely, with the probability α specified upon the creation of the PAA — the key will not result present in the PAA's SBF, and the rule set of the VFDT will not be queried. In this case, we recall that AutoPlacer would consider the key not relocated, and locate it using the default consistent hash function. On other hand, had the SBF suffered of a false positive, the PAA would have used the VFDT's rule set to determine the value associated with k, and also suffer of a false positive.

An example of inaccurate mapping for a key stored in the PAA would have occurred in case the same rule set had been produced with an input seed map containing also the key-value pair $< Customer : SomeName - W1 - D2, 2 >$. In this case the rule set induced by the VDFT algorithm misclassifies the key based on the fact that the value of its Warehouse feature is 1, and erroneously associates it with the target value/node 1, instead than 2.

5 Optimizer Analysis

As already noted, the approximate nature of the information provided by top-k may affect the quality of the identified solution. An interesting question is therefore how degraded is the quality of the data placement solution when using top-k. In the following we provide an answer to this question by deriving an upper bound on the approximation ratio of the proposed algorithm in a single optimization round. Our proof shows that the approximation ratio is a function of the maximum approximation error provided by any $top\text{-}k_j(\mathcal{O})$, which we denote e^*, and of the average frequency of access to remote data items when using the optimal solution.

Theorem 1 The approximation ratio of the solution $\hat{X}$ found using the approximate frequencies $\hat{r}_{ik}$,$\hat{w}_{ik}$ is:

$$1 + \frac{d}{|\mathcal{N}| - d}\phi, \text{ with } \phi = \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW}$$

where e^* is the maximum overestimation error of top-k, and rR, resp. rW, is the average, across all nodes, of the number of read, resp. write, remote data items using the optimal data placement X^{Opt}.

Proof Let us now denote with $C(X, r_{ij}, w_{ij})$ the cost function used in Eq. 1 of the ILP formulation restricted to the data items contained in $top\text{-}K(\mathcal{O})$:

$$\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{O})} \overline{X}_{ij}(cr^r r_{ij} + cr^w w_{ij}) + X_{ij}(cl^r r_{ij} + cl^w w_{ij})$$

and with $Opt = C(X^{opt}, r_{ij}, w_{ij})$ the value returned by the cost function using the binary matrix X^{opt} obtained solving the ILP problem with exact access statistics r_{ij}, w_{ij}.

Let lR, resp. rR, be the average, across all nodes, of the number of read accesses to local, resp. remote, data items using the optimal data placement X. lW and rW are analogously defined for write accesses. These can be directly computed, once known X^{Opt} and r_{ij},w_{ij} as:

$$rR = \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{O})} \overline{X}_{ij}^{Opt} r_{ij}}{|top\text{-}K(\mathcal{O})|(|\mathcal{N}| - d)}$$
$$rW = \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{O})} \overline{X}_{ij}^{Opt} w_{ij}}{|top\text{-}K(\mathcal{O})|(|\mathcal{N}| - d)}$$
$$lR = \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{O})} X_{ij}^{Opt} r_{ij}}{|top\text{-}K(\mathcal{O})|d}$$
$$lW = \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top\text{-}K(\mathcal{O})} X_{ij}^{Opt} w_{ij}}{|top\text{-}K(\mathcal{O})|d}$$

We can then rewrite Opt and derive its lower bound:

$$\begin{aligned} Opt =& |top\text{-}K(\mathcal{O})|(((|\mathcal{N}| - d)(cr^r rR + cr^w rW) + \\ &+ d(cl^r lR + cr^w lW)) \geq \\ &\geq |top\text{-}K(\mathcal{O})|(|\mathcal{N}| - d)(cr^r rR + cr^w rW) \end{aligned} \quad (2)$$

Next, let us derive an upper bound on the "error" using the solution $\hat{X}$ obtained instantiating the ILP problem using the top-k-based frequencies $\hat{r}_{ij}, \hat{w}_{ij}$. The worst scenario is that an object $o \in \mathcal{O}$ is not assigned to the d nodes that access it most frequently because they do not include o in their top-k. In this case we can estimate the maximum frequency with which o can have been accessed by any of these nodes as e^*. Hence if we evaluate the cost function $C(\hat{X}, r_{ij}, w_{ij})$ using the exact data access frequencies, and the solution $\hat{X}$ of the ILP problem computed using approximate access frequencies, we can derive the following upper bound:

$$C(\hat{X}, r_{ij}, w_{ij}) \leq Opt + |top\text{-}K(\mathcal{O})| de^*(cr^r + cr^w) \tag{3}$$

The approximation ratio is therefore:

$$\frac{C(\hat{X}, r_{ij}, w_{ij})}{C(X^{Opt}, r_{ij}, w_{ij})} \leq 1 + \frac{d}{(|\mathcal{N}| - d)} \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW} \tag{4}$$

In the following corollary we exploit the bounds on the space complexity of the Space-Saving Top-k algorithm [35] to estimate the number of distinct counters to use to achieve a target approximation factor $1 + \frac{d}{|\mathcal{N}|-d}\phi$.

Corollary 2 The number m of individual counters maintained by the Space-Saving Top-k algorithm, to achieve an approximation factor equal to $1 + \frac{d}{|\mathcal{N}|-d}\phi$ is:

$$m = \frac{SL}{\phi} \frac{cr^r rR + cr^w rW}{cr^r + cr^w}$$

where SL is the total number of accesses in the stream.

Proof Derives from Theorem 6 of the work that introduced the Space-Saving Top-k algorithm [35], which proves that to guarantee that the maximum overestimation error $e^* \leq \epsilon F_k$, where F_k is the frequency of the k-th element in top-k, it is sufficient to use $m = \frac{SL}{\epsilon F_k}$ counters.

Finally, since in each round AutoPlacer optimizes the placement of a disjoint set of items, it follows that, if we assume stable data access distributions, the approximation ratio achieved by the optimization algorithm during round i will necessarily be lower (hence better) than for round $i - 1$. In fact, at each round, the frequencies of the items tracked by the top-k will be lower than in the previous rounds, and, consequently, e^* will not increase over time.

6 Evaluation

This section evaluates the different aspects of the AutoPlacer. First we describe our experimental setting. Next we discuss the efficiency of the PAA data structure. We then present two case-studies of our system, one based on the well-known OLTP benchmark, TPC-C, and one based on a benchmark representative of geo-social applications, GeoGraph [50]. In order to use both benchmarks in AutoPlacer, we modified the keys of objects in the benchmarks in order to implement the Feature Extractor Interface according to the static attributes of the objects they represent. For both benchmarks, we configured AutoPlacer to run new rounds every minute, and limited the Top-K size to 1000.

6.1 Experimental Settings

In order to evaluate experimentally AutoPlacer, we integrated it in the Infinispan key-value store. We have run experiments on two settings, one for each benchmark we make use of. For the TPC-C benchmark, we have used a cluster of 40 virtual machines (deployed on 10 physical machines) running Xen, equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Ubuntu Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet. For the GeoGraph benchmark, we deployed our platform on of FutureGrid[1]. More specifically, we deployed the Cloud-TM Platform on top of a virtualized infrastructure with 25 VMs, which were configured with 16GB RAM, one 2.93GHz core Intel Xeon CPU X5570, running CentOS 5.5 x86 64 and interconnected by an InfiniBand network.

[1]http://portal.futuregrid.org

Table 3: Re-located objects and size of different PAA implementations

Mechanism	Re-located objects	Local space (KB)
PAA-scalable	26600	150.8
PAA-Bloom	26600	31.84
Bloomier	26600	575.3

TPC-C Benchmark: Since Infinispan provides support for transactions, we developed for our experimental study a porting of a well-known benchmark for transactional systems, namely the TPC-C benchmark [30], which we adapted to execute on a key-value store[2]. This choice is motivated by the fact that TPC-C is a complex benchmark, which generates workloads representative of realistic OLTP environments, with complex and heterogeneous transactions having very skewed access patterns and high conflict probability. This is in contrast with common key-value store benchmarks [13], which are typically composed of simple synthetic workloads.

Since our evaluation focuses on assessing the effectiveness of AutoPlacer in different scenarios of locality, we have modified the benchmark such that we can induce controlled locality patterns in the data accesses of each node. This modification consists in configuring the benchmark such that the transactions originated on a given node access with probability p data associated with a given warehouse, and with probability $1 - p$ data associated a warehouse chosen randomly. So, for example, by setting $p = 90\%$, nodes will have disjoint data access patterns (each accessing a different warehouse) for 90% of the transactions, while the remaining 10% access data uniformly. In the remaining of the text, when we refer to "degree of locality" in the context of the TCP-C benchmark, we refer to the value of probability p above.

GeoGraph Benchmark: Geograph [50] is a benchmarking tool that allows injecting complex and rich workloads representative of the most popular geo-social applications. Geograph has been designed to be flexible both in the heterogeneity and in the dynamics of the workload. In particular, it provides 19 different geo-social services (i.e. actions) and 16 application specific user simulators. These simulators (named agents in GeoGraph) can be combined in complex ways to generate dynamic workload profiles. In order to have a realistic geographical distribution of the agents, GeoGraph uses as baselines for each simulation real GPS traces from Open Street Maps[3].

More specifically, the benchmark divides the spatial region where the agents move into $10KM^2$ areas named Landmarks. It starts by inserting a set of posts (virtual messages) associated with each Landmark. Then, agents are spread throughout the map and move across it using the realistic GPS traces, while issuing actions in its vicinities. The simplest action is read-post action which is used to query the landmark state for getting posted messages. The second action is update-position which leads to a move operation of the agent between adjacent landmarks, and possibly to leave a new post.

In order to simulate a real-work deployment of a geo-social service, we have configured the benchmark to dispatch requests across the nodes in the system depending on their geographical origin. We then attributed geographical areas (sets of adjacent Landmarks) to servers, such that the requests of an agent are dispatched to the server responsible for the closest landmark, based on the current geographic position of the agent.

6.2 Probabilistic Associative Array

In this section we study tradeoffs in the space efficiency and accuracy involved in the configuration and implementation of the PAA. For these results, we have used traces of the TPC-C benchmark configured with 100% locality.

[2]The code of AutoPlacer and of the porting of TPC-C employed in this evaluation study are freely available in the Cloud-TM project public repository: http://github.com/cloudtm

[3]http://www.openstreetmap.org/traces

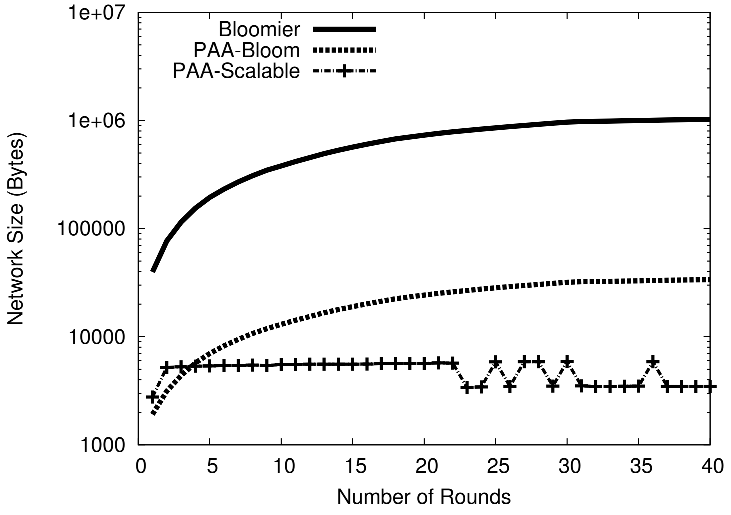


Figure 2: Traffic generated by a node using different associative arrays.

Bloom Filter Figure 2 presents the network bandwidth of different implementations of the PAA, compared with another form of probabilistic associative arrays, the Bloomier Filters (Bloomier) [9] as the rounds advance in the system. One implementation of the PAA uses regular Bloom filters (PAA-Bloom) [6], while the other uses a scalable Bloom filter (PAA-scalable) [1]. Both PAAs were configured with $\alpha = \beta = 0.01$, and the Bloomier filter's false positive probability was also set to 0.01. We note that the best solution is the one that allows to propagate in the network only differential updates with regard to the previous state, i.e, the PAA-scalable. Concerning this latter solution, Figure 2 shows that in some rounds the message traffic generate by this solution oscillates. This can be explained considering that, as a result of the insertion of new data in the PAA, it may be possible to exhaust the capacity of the most recent internal BF of the scalable Bloom filter, and trigger the allocation of a new slice. In this case, the getDelta method of the PAA returns the last two (or potentially more, although this never occurred in this experiment) internal BFs.

Table 3 shows the correspondent local storage requirements at the end of the experiment. As it can be seen, PAA-scalable has higher storage requirements than PAA-Bloom. This is unsurprising, as scalable bloom filters are known to achieve lower compression than traditional bloom filters when fed with the same data set and configured to yield the same false positive rates [1]. However, the storage requirements of both solutions are still considerably smaller than those of Bloomier filters.

Machine Learner Figure 3 presents the error probability and space required by the DT to encode the objects moved in every round of the experiment. As more objects are moved in the system, the number of rules increases, leading to an increase in the size taken by this portion of the PAA. However, the machine learner can represent the mapping of 26600 keys in 1000 Bytes, which correspond to 213 rules. Furthermore, it can also be observed that while a significant number of keys is added to the machine learner (around 1000 per round), the error remains relatively stable.

6.3 Leveraging from Locality

This section shows how AutoPlacer is able to leverage form locality patterns in the workload. The results were obtained with TPC-C, adapted as explained before and with a replication of degree $d = 2$.

Figure 4(a) shows the throughput of AutoPlacer, compared with the non-optimized system using consistent hashing for different degrees of locality in the workload. In the baseline system, no matter how much locality exists in the workload, since consistent hashing is used to place the items, on average the number of remote accesses does not change. Thus, for all workloads the baseline system exhibits a similar (sub-optimal) behaviour. In the system running AutoPlacer, locality is leveraged by relocating data items.

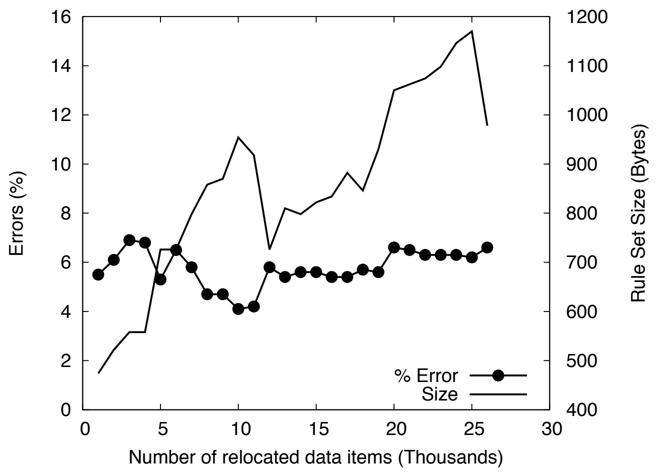
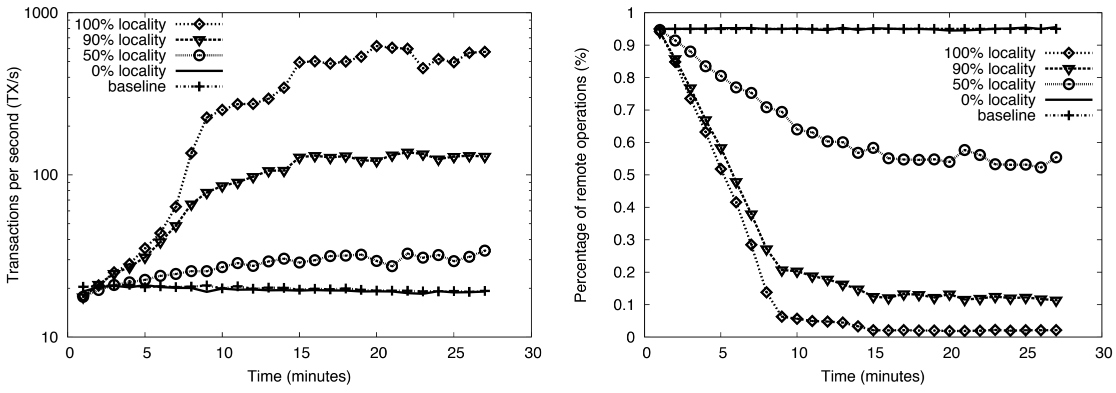


Figure 3: Error probability and rule set size for VFDT



(a) Throughput with varying degrees of locality.

(b) % of remote operations.

Figure 4: AutoPlacer performance

As times passes, and more rounds of optimization take place, the system throughput increases up to a point where no further optimization is performed. It is interesting to note that, in case there is no locality, the throughput is not affected by the background optimization process. On the other hand, when locality exists, the throughput of the system optimized with AutoPlacer is much higher than that of the baseline: it can be up to 6 times better for a workload with 90% locality, and up to 30 times better in the ideal case of 100% locality.

Figure 4(b) helps to understand the improvement in performance by looking at the number of remote invocations that are performed in the system as time evolves. Since the initial setup relies on consistent hashing, both in the baseline and in the optimized system, the average probability of an operation being local is $\frac{1}{40} = 2.5\%$ for all workloads. Thus, when the system starts most operations are remote. However, the plots clearly show that the number of remote operations decreases in time when using AutoPlacer. The plots also show another interesting aspect: although the number of remote operations decreases sharply after a few rounds of optimization, the overall throughput takes longer to improve. This is due to the fact that read transactions access a large number of objects, thus multiple rounds of optimization are required to alleviate the network, which is the bottleneck in these settings. This clearly highlights the relevance

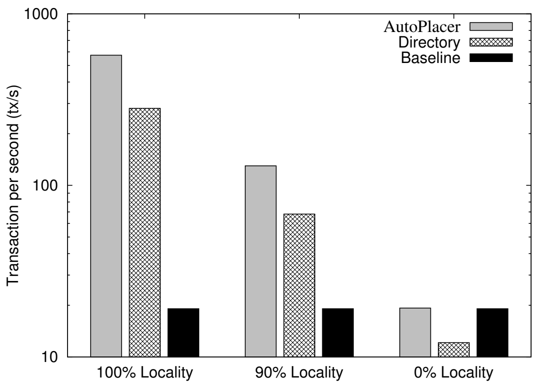


Figure 5: Throughput of AutoPlacer, a directory-based and a consistent hashing-based solution, after a complete optimization process.

of the continuous optimization process implemented by AutoPlacer. At the end of the experiment, the percentage of operations performed locally is already close to the percentage of locality in the workload; this shows that when the system stabilizes, AutoPlacer was able to move practically all keys subject to locality.

Finally, Figure 5 compares the performance of AutoPlacer and that of a directory service-based system which may be used to store the mappings resultant from a global optimization in systems such as Ursa [48] or Schism [16] among other state of the art solutions which lack an efficient way of broadcasting mappings. These results were obtained by storing the data relocation map obtained at the end of the entire optimization process into a dedicated directory service. In this case, whenever a node requests a data item that is not stored locally, it contacts the directory service to determine its location, instead of querying the local PAA. The results clearly show that the additional latency for contacting the directory service can hinder perform significantly, independently of the locality of the workload. The plots highlight that, unlike for AutoPlacer, the performance of directory-based systems can be worse than that achievable by using random placement. This is explainable considering that, with low locality, a large fraction of data accesses is remote, and that directory-based services impose a 2-hop latency, unlike consistent hashing and AutoPlacer. Note that, while the plot depicts throughput values, these numbers are highly correlated with the latency of individual operations; in fact, the decrease of throughput from AutoPlacer to that of a directory-based system is due to the latency of lookups and the decrease of throughput from the directory-based system to the baseline is due to the latency of (non-local) data accesses.

The speed-ups of AutoPlacer vs the directory-based solution are significant, i.e. around 2x, even for the high locality scenarios. In these scenarios, the reduction of the number of remote operations leads to less lookups on the directory. However, the cost of accessing a remote data item is, in our testbed, about 2 orders of magnitude larger than that of accessing a local item. As a consequence, also in these scenarios, the cost of remote data accesses dominate the execution time of the requests. Hence, such requests, which suffer from one additional communication hop latency in a directory-based solution, effectively limit the throughput of such a solution leading to considerably worse results than AutoPlacer.

6.4 Distributed Optimization

In this section we explore how the design decisions behind AutoPlacer's distributed optimization algorithm affect the optimality of the final data placement. We have implemented a version of our optimizer which works in a centralized way using complete and precise traces of the system, obtained from all nodes during a long period. This is an approach similar to that of state of the art solutions such as Ursa [48] and Schism [16]. Finally, in order to improve the quality of the solution of the optimizer, such that it becomes an optimal adversary, we did not relax the ILP constraint on this version.

To achieve a fair comparison, we did three runs of TPC-C with the same settings as Section 6.3, and with two different configurations of Infinispan. Firstly, we did two long runs of an unmodified Infinispan,

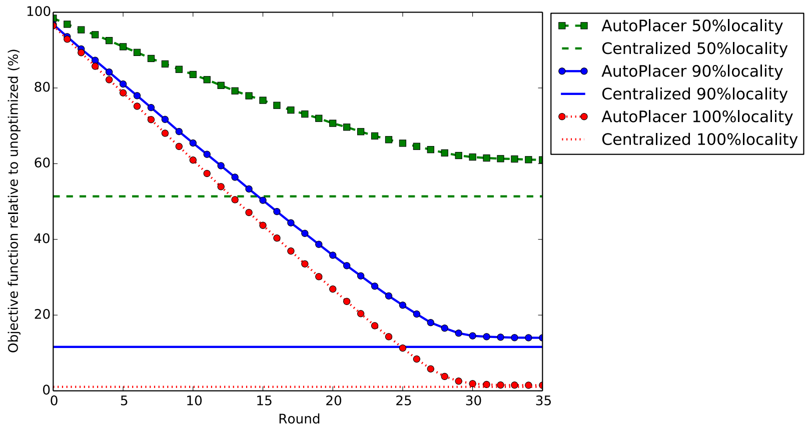


Figure 6: Progression of the quality of the data placement optimization solutions of AutoPlacer's distributed optimization and of a centralized optimizer, relative to an unoptimized placement

while collecting traces of data access. We used the traces from the first run to generate an optimization problem, which we fed to the centralized optimizer to derive an optimized placement. Next, we used the traces from the second run to generate another optimization problem, and store only its cost function. Finally, we ran AutoPlacer for the same period and collected the data placements resultant from each round of optimization. In this evaluation, we use the traces from the second run as a fair baseline for comparison between AutoPlacer and a centralized optimizer; We apply the data placement resultant from AutoPlacer and from the centralized optimizer to the cost function derived from the traces of the second run. This allows us to compare the quality of the solution returned by each optimizer with respect to an independent run from that which the optimizers used as input.

Figure 6 shows how the quality of the solution generated by AutoPlacer evolves along rounds, and the quality of the solution generated by the centralized optimizer. This quality is a result of applying the cost function obtained from the second Infinispan run to the data placement solutions generated by each optimizer. The results are presented as a percentage of the cost of the (unoptimized) data placement generated by consistent hashing.

Similarly to the behaviour observed in Section 6.3, AutoPlacer converges progressively towards a steady-state, where the value for the objective function is not reduced any further. As expected, the more locality is encoded in the workload, the smaller is the value at which AutoPlacer converges, since more requests can be fulfilled by moving replicas to nodes requesting data items. In fact, the results shown in Figure 6 match those of Figure 4(b): since the remote operations are considerably more expensive than the local operations, the value of the objective function at the end of the experiment relative to the unoptimized version corresponds roughly to the percentage of remote operations relative to the baseline in Figure 4(b).

Regarding the comparison with the centralized optimizer, it can be observed that AutoPlacer's optimizer's results are hindered by the fact that it is distributed and relies on uncertain information. However, it must be noticed that AutoPlacer optimizes the placement based on a much shorter window of time, which means that the statistics are more prone to be unreliable. This fact, combined with the fact that AutoPlacer avoids moving data more than once on each round, means that the system may make more wrong decisions. In absolute terms, this difference is more pronounced for lower locality scenarios, since AutoPlacer can more easily be deceived by the randomness in the workload than an algorithm which decides placement based on a longer period. In relative terms, the difference is more pronounced for higher locality scenarios, since any single deviation from the data placement devised by the centralized optimizer will have a strong impact on the objective function. Still, AutoPlacer was able to approach the results of the centralized optimizer to within 10% of the cost of the unoptimized placement, without requiring a complete trace of execution and while allowing the parallelization of the computation. Furthermore, AutoPlacer also provided considerable gains over the unoptimized placement, especially for the high locality scenarios where the cost is reduced by up to 98.5%.

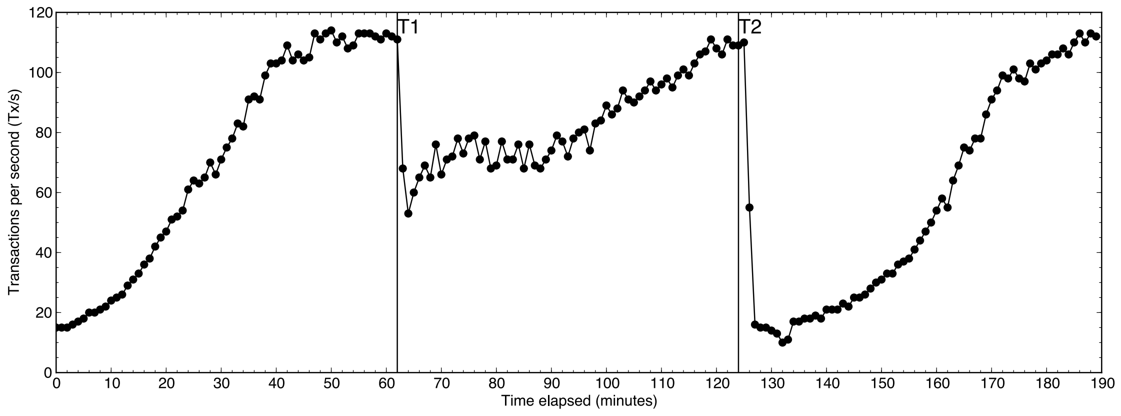


Figure 7: Total throughput of AutoPlacer over time for a dynamic workload with 90% of locality. T1 and T2 mark the instants when changes in workload occurred.

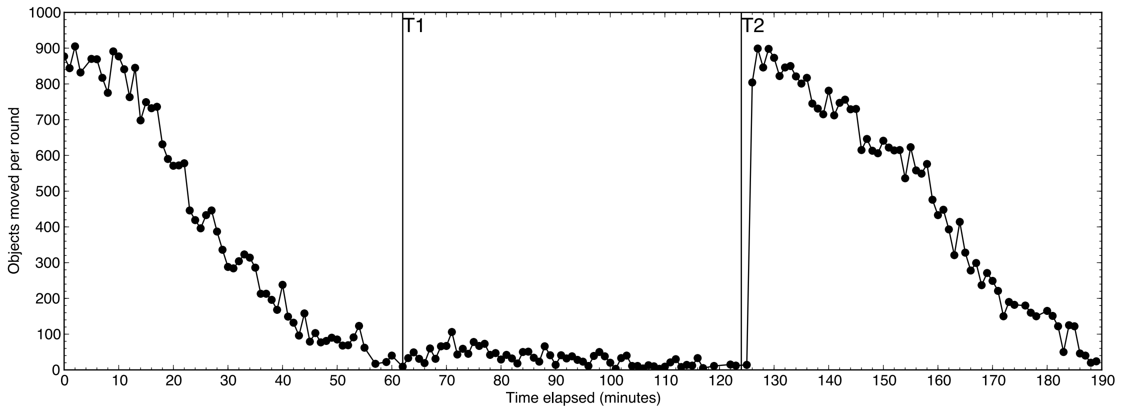


Figure 8: Number of objects moved to a specific node per round over time. Figure presents data for one of the nodes whose data is not affected by the first workload change (at T1).

6.5 Dynamic workloads

In this section we show how changes in the workload affect the behaviour of AutoPlacer. We deployed TPC-C using the same settings as in Section 6.3, with 90% access locality, but modified the benchmark to be able to introduce changes to the data access pattern at runtime. This allows us to emulate the existence of a load balancer which may at any time decide to change the way in which requests are routed to each node (e.g., due to the change of popularity of some subset of data items).

Figure 7 presents the evolution of the throughput of the system when subject to two changes in workload, taking place at time T1 and T2. At T1, we cause a change in the data access pattern of 50% of the nodes. This change causes these nodes to access a different set of data, and to start access data which was previously being accessed by some other node in the set. At T2, we perform a similar change, to alter the data access pattern for all nodes in the system.

As Figure 7 shows, the throughput of the system drops considerably at both T1 and T2. This is caused by an increase in the remote read operations in the nodes affected by the change, which not only cause their transactions to last longer, but also flood the network, affecting the transactions of the other nodes. It is noticeable, however, that the throughput drop at T1 is considerably smaller than at T2, since only 50% of the nodes are affected by the change. Furthermore, after T2 the throughput drops to roughly the same as at the beginning of the experiment, since after the change at T1 most of the read operations become remote as at the beginning of the experiment.

In terms of convergence time, each section of the experiment runs for 64 minutes, but after roughly 40 minutes no significant increases in throughput are observed. It is important to notice that this fact also holds for the second section of the experiment (between T1 and T2). Even though the drop in performance after T1 is not as large as after T2, AutoPlacer still takes the same time to converge on the peak throughput as for the other sections. This happens because at each round, each node moves a limited number of objects, hence it will always require a similar number of rounds to optimize a the same number of objects.

Figure 8 shows the number of objects moved to one of the nodes not affected by the change in workload triggered in T1. From 0 to T1, the node receives most of its objects in the first rounds of optimization, and as the system reaches a more stable throughput the number of objects moved is reduced. Notice also that even though the node still receives around 80 objects per round after the throughput has stabilized, this object relocation leads to a marginal throughput gain. Between T1 and T2, this node's data remains roughly the same: since it is not affected by the workload change, it does not request any data from its neighbours. After T2, a new epoch is triggered, and since the node considered in Figure 8 is affected by the workload change, it starts requesting and moving data to itself, until the system reaches a stable state towards the end of the experiment.

6.6 GeoGraph evaluation

This section studies the behaviour of AutoPlacer when using GeoGraph. Figure 9 presents the results for AutoPlacer running the GeoGraph benchmark. We started using the default, consistent-hashing-based data placement, and disabling AutoPlacer. After 10 minutes we activated AutoPlacer's statistics collection. After a 10 minutes period to warm up the statistics, at minute 20 we triggered the first optimization epoch.

The plots in the top row of Figure 9 show how throughput and response time of read-only transactions vary over time during an experiment in which we inject load using 128 geographically dispersed agents, generating 90% read-only transactions (i.e., 90% read-post operations and 10% update-position operations) with 0 think time. After approximately 10 minutes, when the performance of the system is of about 60 transactions per second, we trigger the gathering of statistics using the probabilistic top-k algorithm integrated in AutoPlacer. We notice that the throughput has a temporary drop of performance, with the throughput going down to 53 transactions over the following sampling interval, before achieving a stable throughput of about 57 transactions per second after about 5 minutes. The temporary drop in performance, immediately after the activation of the Top-K tracing is most likely imputable to the initial overheads of the JVM in executing new code paths before the JIT targets them and optimizes them. Interestingly, however, the performance penalty introduced by top-k fades away almost completely over the following 5 minutes, where the system stabilizes its performance at about 5% lower than without Top-K tracing.

By looking at the plots concerning the average number of remote get operations for both write and read-only transactions, in the bottom row of Figure 9, we can notice that transactions are generating a large amount of remote accesses, resp. about 23 for write and about 360 for read-only transactions. This is a clear symptom of poor application's locality, which represents a typical use case that could potentially benefit significantly from AutoPlacer's locality enhancing data-placement scheme.

We trigger AutoPlacer at around minute 20, and configure it to execute 10 rounds, collecting in each round statistics for 1000 keys per node. In order to assess the benefits on performance associated with the activation of each individual round we set a conservative frequency of 1 activation each 10 minutes. This choice allows to have sufficient time to observe the dynamics of response of the system to the activation of the individual AutoPlacer's rounds. Analogously to the results obtained when using TPC-C, AutoPlacer provides significant benefits in terms of locality of the applications' data access patterns: over time we see that the number of remote operations issued during the execution of transactions drops considerably, with the first rounds being the most effective. This phenomenon is explicable considering that during the early optimization rounds AutoPlacer relocates the hot-spot data items that are responsible for causing most of the remote accesses in each node, whereas in the later rounds the objects appearing in the nodes' top-k statistics are going to be accessed less and less frequently by the nodes. The last round ends at about 120 minutes since the beginning of the experiment.

At the end of the experiment we obtain that the throughput fluctuates around 220 transactions per second, yielding a throughput gain of about 4 times. As expected, the throughput of AutoPlacer clearly increases

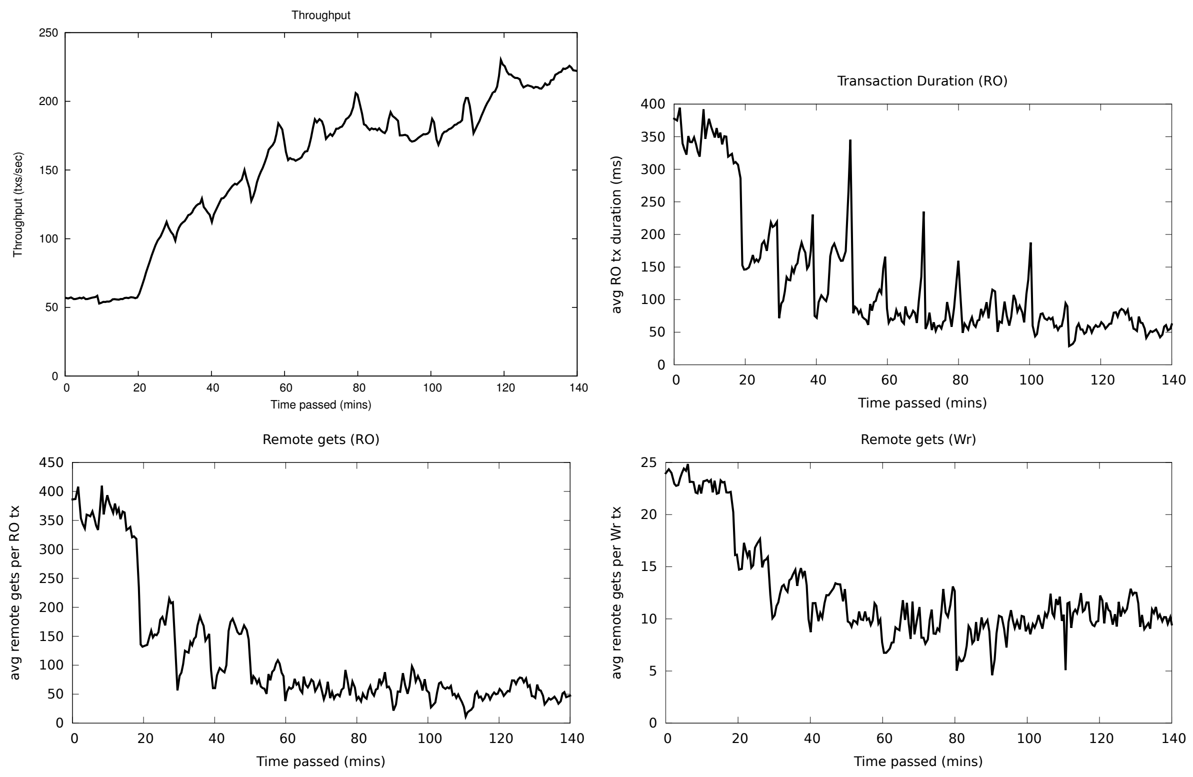


Figure 9: Evolution over time of throughput, duration of read-only transactions, and number of remote get operations per read-only and update transactions – GeoGraph benchmark.

over time, peaking at more than $4x$ of the initial, non-optimized throughput. This result is particularly good since AutoPlacer can infer the locality patterns of a realistic workload without requiring the programmer to know *a-priori* how to partition the application's data and map it to the set of available nodes. The root causes of these gains are highlighted by the plots concerning the number of remote operations, as well as by the duration of read-only operations: both of them decrease significantly as subsequent rounds of AutoPlacer are executed, which leads to a progressive growth of efficiency of the system.

7 Related Work

A common approach to implementing data placement mechanisms in large scale systems is to rely on coarse-grained, user-defined data partitions/buckets (also named directories [14] or tablets [12, 8]). Through such partitioning, systems can deploy a centralized component which manages the location of all buckets in the system, moving them as required to balance the load on hotspot nodes. While coarse partitioning allows for somewhat manageable directories (maintaining the mapping between objects and nodes), on the other hand, its coarse granularity can reduce the effectiveness of the load balancing mechanisms. Furthermore, to improve data locality, these systems make use of sorted keys: the programmer is responsible for assigning similar keys to related data in order for it to be placed in the same server (or in the same group of servers) [14, 12, 27]. AutoPlacer does not require the programmer to manually define data partitions. While AutoPlacer exploits information encoded in the key structure, the programmer is not required to define partitioning rules. Conversely, these are automatically inferred using machine learning techniques based on the data access patterns exhibited by the various nodes in the system. Also, by inferring data partitioning rules using an on-line decision tree algorithm, our system can establish a fine grain placement for the most accessed items in a space-efficient way.

As hinted several times in the paper, there is extensive work on defining optimal data placement strategies in multiple contexts. A naive design that achieves an optimal object-to-node mapping is to rely on a centralized component. In such a scheme, nodes send their object request to a single node, which runs an optimization algorithm based on some variant of the File Allocation Problem (FAP) [7, 29, 49, 28], and then the new mappings are propagated to a router component which creates lookup tables [44] and routes requests to system nodes. However, not only does this solution involve an extra network hop for each request, but also FAP is an NP-complete problem (hence inherently non-scalable in terms of the number of objects) and this design would cause nodes to send information on every single object they store to a centralized location. Finally, [29] shows that such an algorithm can react slowly to changes in workloads since the decisions are not local to the nodes. Despite these drawbacks, several state-of-the art works follow approaches inspired on this naive solution: [24] and [15] use a centralized component which collects fine-grained statistics on object usage, derives a placement using an optimization algorithm for improving load balancing across the system and then acts as a directory to locate objects. Ursa [48] follows a similar design also for achieving load balancing goals, but it only considers the most used objects in order to improve scalability, in a similar way as AutoPlacer uses top-k. Schism [16] and the work by Pavlo *et al.* [37] also follow this centralized design, with a similar goal as AutoPlacer of improving data locality. To achieve optimal placement, these latter two systems collect system execution logs to extract statistics on correlated objects, and then use graph partitioning algorithms to place the objects into groups which are then mapped to nodes. Despite achieving near-optimal placements, these are mostly offline algorithms, which consider all data in the system and have a limited scalability as the number of data items grows.

The work by Vilaça *et al.* [45] presents a Space-Filling Curves-based approach to placing co-related data in the same nodes by relying on user-defined per-object tags. The resulting system can provide good locality if the application is designed to perform actions using the tags, since nodes can locally determine who are the owners of the objects with specific tags. However, unlike our system, this placement is static and encoded by the programmer, and has no relation with the actual data access patterns that may emerge at runtime.

The recent work in [32] proposes Proteus, a virtual node placement algorithm that allows minimizing the delay penalties otherwise incurred during server provisioning dynamics when using a conventional consistent hashing placement scheme. This scheme could be easily integrated with AutoPlacer, as a mechanism like Proteus could be used in AutoPlacer as alternative to the classic consistent hashing scheme.

Finally, this work is also related to the vast literature on self-management [11, 22] of cloud data platforms, and in particular to the problems of thermal optimization [31, 10] and SLA-based provisioning [46, 18]. The self-tuning mechanisms designed to cope with these problems can induce significant variations of the locality patterns exhibited by the nodes of a distributed key value store, e.g., by altering dynamically the amount of allocated resources, or the way in which requests are dispatched to resources. AutoPlacer can be used to ensure that, whenever these systems reconfigure the platform to meet SLA or thermal constraints, the placement of data across the nodes of the system is always constantly optimized.

8 Conclusions

This paper presented AutoPlacer, a system aimed at self-tuning data placement in a distributed key value store. AutoPlacer operates in rounds, and, in each round, it optimizes the placement of the top-k "hotspots", i.e. the objects generating most remote operations, for each node of the system. Despite supporting fine-grain placement of data items, AutoPlacer guarantees one hop routing latency using a novel probabilistic data structure, the PAA, which minimizes the cost of maintaining and disseminating the data relocation map. AutoPlacer has been integrated in a popular open source (transactional) key-value store, Infinispan, and experimentally evaluated using a porting of the TPC-C benchmark. The results shown that AutoPlacer can achieve a throughput up to six times better than the original Infinispan implementation based on consistent hashing.

References

[1] Almeida, P., Baquero, C., Preguiça, N., and Hutchison, D. Scalable bloom filters. *Inf. Process. Lett.* (Mar. 2007).

[2] Amza, C., Cox, A., and Zwaenepoel, W. Conflict-aware scheduling for dynamic content applications. In *Proc. of the 4th USITS* (Seattle (WA), USA, 2003).

[3] Ban, B., et al. Jgroups, a toolkit for reliable multicast communication.

[4] Bauer, C., and King, G. *Java Persistence with Hibernate.* Manning Publications Co., Greenwich, CT, USA, 2006.

[5] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] Bloom, B. Space/ time trade-offs in hash coding with allowable errors. *Comm. of the ACM* (1970).

[7] Chandy, K. M., and Hewes, J. E. File allocation in distributed systems. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation* (New York, NY, USA, 1976), SIGMETRICS '76, ACM, pp. 10–13.

[8] Chang, F., et al. Bigtable: a distributed storage system for structured data. In *Proc. of the 7th OSDI* (Seattle, USA, 2006).

[9] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. of the 15th SODA* (New Orleans (LA), USA, 2004).

[10] Chen, H., Song, M., Song, J., Gavrilovska, A., and Schwan, K. Hears: A hierarchical energy-aware resource scheduler for virtualized data centers. In *2011 IEEE International Conference on Cluster Computing* (2011), CLUSTER, pp. 508–512.

[11] Cook, N., Milojicic, D. S., and Talwar, V. Cloud management. *Journal Internet Services and Applications 3*, 1 (2012), 67–75.

[12] Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. In *Proc. of the 34th VLDB* (Auckland, New Zealand, Aug. 2008).

[13] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st SoCC* (New York, NY, USA, 2010).

[14] Corbett, J., et al. Spanner: Google's globally-distributed database. In *Proc. of the 10th OSDI* (Hollywood, CA, USA, 2012).

[15] Cruz, F., Maia, F., Matos, M., Oliveira, R., Paulo, J. a., Pereira, J., and Vilaça, R. Met: Workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 183–196.

[16] Curino, C., Jones, E., Zhang, Y., and Madden, S. Schism: a workload-driven approach to database replication and partitioning. In *Proc. of the 36th VLDB* (Singapore, Sept. 2010).

[17] DeCandia, G., et al. Dynamo: amazon's highly available key-value store. In *Proc. of the 21st SOSP* (Stevenson, USA, 2007).

[18] Didona, D., Romano, P., Peluso, S., and Quaglia, F. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing* (2012), ICAC '12, ACM, pp. 125–134.

[19] Domingos, P., and Hulten, G. Mining high-speed data streams. In *Proc. of the 6th KDD* (Boston, MA, USA, 2000).

[20] Dowdy, L., and Foster, D. Comparative models of the file assignment problem. *ACM Computing Surveys* (1982).

[21] Fleisch, B., and Popek, G. Mirage: a coherent distributed shared memory design. *SIGOPS Oper. Syst. Rev.* (Nov. 1989).

[22] Forell, T., Milojicic, D. S., and Talwar, V. Cloud management: Challenges and opportunities. In *IPDPS Workshops* (2011), pp. 881–889.

[23] Garbatov, S., and Cachopo, J. Data access pattern analysis and prediction for object-oriented applications. *INFOCOMP Journal of Computer Science* (December 2011).

[24] Jia, Y., Brondino, I., Peris, R. J., Martínez, M. P. n., and Ma, D. A multi-resource load balancing algorithm for cloud cache systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (2013), SAC '13, ACM.

[25] Jiménez-Peris, R., Patiño Martínez, M., and Alonso, G. Non-intrusive, parallel recovery of replicated data. In *Proc. of the 21st IEEE SRDS* (Washington, DC, USA, 2002).

[26] Krishnan, P., Raz, D., and Shavitt, Y. The cache location problem. *IEEE/ACM Transactions on Networking* (Oct. 2000).

[27] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* (Apr. 2010).

[28] Laoutaris, N., Telelis, O., Zissimopoulos, V., and Stavrakakis, I. Distributed selfish replication. *IEEE TPDS* (Dec. 2006).

[29] Leff, A., Wolf, J., and Yu, P. Replication algorithms in a remote caching architecture. *IEEE TPDS* (Nov. 1993).

[30] Leutenegger, S., and Dias, D. A modeling study of the tpc-c benchmark. In *Proc. of the SIGMOD Conf.* (Washington, D.C., United States, 1993).

[31] Li, S., Abdelzaher, T., and Yuan, M. Tapa: Temperature aware power allocation in data center with map-reduce. In *Green Computing Conference and Workshops (IGCC), 2011 International* (2011), pp. 1–8.

[32] Li, S., Wang, S., Yang, F., Hu, S., Saremi, F., and Abdelzaher, T. F. Proteus: Power proportional memory cache cluster in data centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems* (2013), ICDCS, pp. 73–82.

[33] Liu, H., and Motoda, H. *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[34] Marchioni, F., and Surtani, M. *Infinispan Data Grid Platform.* PACKT Publishing, 2012.

[35] Metwally, A., Agrawal, D., and El Abbadi, A. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th ICDT* (Edinburgh,Scotland, 2005).

[36] Mitchell, T. *Machine Learning.* McGraw-Hill, New York, 1997.

[37] Pavlo, A., Curino, C., and Zdonik, S. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 61–72.

[38] Peluso, S., Romano, P., and Quaglia, F. Score: A scalable one-copy serializable partial replication protocol. In *Proc. of the 13th Middleware* (2012), pp. 456–475.

[39] Peluso, S., Ruivo, P., Romano, P., Quaglia, F., and Rodrigues, L. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proc. of the 32nd ICDCS* (2012), pp. 455–465.

[40] Quinlan, J. R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[41] Romano, P., Little, M., Quaglia, F., Rodrigues, L., and Ziparo, V. A. Cloud-tm: Transactional, object-oriented, self-tuning cloud data store. Tech. Rep. 7, INESC-ID, April 2014.

[42] Ruivo, P., Couceiro, M., Romano, P., and Rodrigues, L. Exploiting total order multicast in weakly consistent transactional caches. In *Proc. of the the 17th PRDC* (Pasadena, California, USA, Dec. 2011).

[43] Stanoi, I., Agrawal, D., and Abbadi, A. E. Using broadcast primitives in replicated databases. In *Proc. of the The 18th ICDCS* (Washington, DC, USA, 1998).

[44] Tatarowicz, A. L., Curino, C., Jones, E. P. C., and Madden, S. Lookup tables: Fine-grained partitioning for distributed databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering* (Washington, DC, USA, 2012), ICDE '12, IEEE Computer Society, pp. 102–113.

[45] Vilaça, R., Oliveira, R., and Pereira, J. A correlation-aware data placement strategy for key-value stores. In *Proc. of the 11th DAIS* (Reykjavik, 2011).

[46] Wang, L., Xu, J., Zhao, M., and Fortes, J. Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA, 2011), ICAC '11, ACM, pp. 191–192.

[47] Witten, I. H., and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[48] You, G.-W., Hwang, S.-W., and Jain, N. Ursa: Scalable load and power management in cloud storage systems. *ACM Transactions on Storage 9*, 1 (Mar. 2013), 1:1–1:29.

[49] Zaman, S., and Grosu, D. A distributed algorithm for the replica placement problem. *IEEE TPDS* (Sept. 2011).

[50] Ziparo, V., F., C., D., C., M., Z., F., G., and P., R. D4.3 - prototype of pilot application i. In *Cloud-TM project* (2013).