

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, Portugal

Abstract—This paper addresses the problem of data placement in replicated key-value stores. The goal is to place data replicas 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 data placement, but also to preserve fast data lookup. The paper introduces new techniques that address each of the challenges above. These techniques have been integrated in a prototype that has been experimentally evaluated. The performance results show that the throughput of the optimized system can be 6 times better than a baseline system using consistent hashing.

I. INTRODUCTION

Distributed NoSQL key-value stores [12], [19] have emerged as the reference architecture for data management in the cloud. A fundamental design choice in these distributed data platforms is related to the algorithm used for determining 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 storing replicas of the data in the nodes that access them more frequently, typically subject to constraints on the object replication degree and on the capacity of the nodes; ii) maximizing look-up speed, by ensuring that a copy of an object can be located as quickly as possible.

The data placement problem has been largely investigated in literature in many alternative variants [14], [17]. 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 making approaches attempting to optimize the placement of each item inherently non-scalable. Even if the optimal placement could be computed, it is challenging to maintain efficiently a (potentially extremely large) directory to store the mapping among items and storage nodes. Existing key-value stores tackle this issue by adopting two main strategies: random placement or dedicated directory services.

Random placement solutions, typically based on consistent hashing [19], are employed by several popular key-value stores [19], [12]. By relying on simple, random hash functions to disseminate data across nodes, these solutions allow lookups to be performed locally, in a very efficient manner [12]. Furthermore, consistent hashing guarantees that the join/leave of a node incurs in a limited change in the mapping of keys to buckets. However, due to the random nature of data placement (oblivious to the access frequencies of the node to data), solutions based on consistent hashing may result in highly

sub-optimal data placements. This can lead to the generation of a large number of remote data accesses, which may slow down, and possibly trash, the system.

Other systems, like PNUTS [9] or BigTable [6], rely on dedicated directory services, which allows to achieve flexibility in the mapping between objects and nodes (typically coarse data partitions rather than on a per instance basis). However, the reliance on remote directory services introduces additional round-trip delays along the critical execution path of data access operations, which can hinder performance considerably.

This paper introduces 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 [9], [6], we 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 the placement of the top- k “hotspots”, i.e. the objects generating most remote operations, for each node of the system. Not only this design choice allows reducing the number of decision variables of the data placement problem (solved at each round) and ensuring its practical viability, it also abates the monitoring overhead for tracking and exchanging data access statistics.

In order to minimize the overhead for identifying the “hotspots” of each node, AUTOPLACER adopts a state of the art stream analysis Space-Saving Top- k algorithm [24] 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 problem’s solution, showing how it can be made amenable for being partitioned in independent subproblems, solvable in parallel.

Unlike solutions that rely on dedicated 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) hot data items.

The key innovative solution introduced to pursue this goal is a novel data structure, which we named *Probabilistic Associative Array* (PAA) and that we use 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 use probabilistic techniques and trade-off accuracy, and can reply erroneously to queries with a user-tunable probability p . 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; IDTs 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 define an intuitive programmatic API that allows programmers to provide additional information on the keys to be stored in the PAA (e.g., the data type of the object 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.

The resulting system has been implemented as a variant of the Infinispan key-value store. Infinispan is the reference NoSQL platform and clustering technology for JBoss AS, a mainstream open source J2EE application server. The results shown that AUTOPLACER can achieve a throughput 50 times better than a baseline system using consistent hashing.

The remaining of the paper is structured as follows. Section II provides a global overview of AUTOPLACER. The following sections, describe the major components of the architecture: hot-spot identification and data placement optimization are described in Section III, and the PAA data structure is described in Section IV. An extensive evaluation of the system is provided in Section VI. Finally Section VII compares our system with related work and Section VIII concludes the paper.

II. SYSTEM OVERVIEW

A. Target System

The development of AUTOPLACER has been motivated by our experience with the use of an existing, state-of-the-art, transactional key-value store, namely the Infinispan product by RedHat[23]. In Infinispan (and other similar products such as [19], [12]), data is stored in multiple nodes. 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.

We denote the node where a transaction is submitted the transaction's *delegate node*. Transactions read objects from their home nodes (which can be other servers from the cluster) and locally buffer writes until commit time. If a transaction successfully commits¹, updates are applied back to the home nodes of the written items. The most efficient scenario occurs when a read only transaction executes in a server and all objects accessed by the transaction are stored locally. In this case, the transaction may commit without engaging with remote communication to other servers. Conversely, the less

favorable case occurs when all data items are stored remotely. A similar reasoning can be made for update transactions.

In Infinispan, data can be replicated. In this case, even if all items are available at the delegate node, an update transaction must always engage in communication (to ensure that all replicas are updated). Hence, the higher the number of nodes that maintain replicas of the data items updated by a write transaction, the larger the overhead associated with the commit phase (in terms of generated network traffic).

Infinispan uses consistent hashing to distribute data items among nodes in a random fashion, ensuring that lookups can be executed locally in a very efficient fashion. Unfortunately, in typical applications of large-scale key-value stores, random data placement can be largely suboptimal as transactions are likely to generate skewed access distributions [22], often dependant on the actual "type" of transactions processed by each node [28], [11]. Also, workloads are frequently distributed according to load balancing strategies that strive to maximize locality [16]/minimize contention [2] in the data accesses generated by each node.

B. The AUTOPLACER Approach

AUTOPLACER executes continuously a sequence of optimization rounds. As a result of each round, a number of data item may be relocated, if the expected gains are above a minimum threshold. Each optimization round consists of the following sequence of six tasks.

The first task consists of collecting statistics about the data accessed by the transactions that execute in each node. This information provides a record of which items generate communication in the cluster and which nodes are accessing those items at a given point in time.

The information collected in the previous task, feeds the second task that consists in identifying *hotspots*, i.e., a small subset of data items that is responsible for the larger fraction of all communication in the cluster. In fact, instead of trying to optimize the placement of every data item, AUTOPLACER only attempts to optimize the placement of items that are identified as hotspots. So, each node collects the statistics for the hotspots for the data items which it stores. Since this task is run repetitively, once some hotspots have been identified (and relocated) in a given round, new (different) hotspots are seek in the next round. Therefore, although in each round only a limited number of hotspots is identified, in the long run, many data items may be selected, as long as gains can still be obtained from their relocation.

Once the hotspots have been collected, the third task consists in finding an appropriate placement for those items. This task is denoted the *optimization* task. The result of this task is a *relocation map*, i.e., a mapping of where replicas of each hotspot item must be placed.

The relocation placement map is used to move data items physically in the cluster and later used by all nodes to lookup the location of hotspot items. However even if hotspots are only a small fraction of the entire set of items maintained in the key-value store, an explicit map can grow in an undesired way, take precious memory, and may even be too large to be efficiently distributed to all nodes. This motivates a fourth task executed by AUTOPLACER, that consists in encoding

¹The coordination required to commit a transaction is irrelevant to this paper; the interested reader may refer to [7].

\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
C_j	the capacity of node j .

Table I
PARAMETERS USED IN THE ILP FORMULATION.

the relocation map in a data structure that can be efficiently communicated in the system and that can also support fast lookups. The encoding task uses as input a relocation map and produces a *Probabilistic Associative Array* (PAA). Once computed the PAA is disseminated among all nodes (this corresponds to the fifth task).

At the end of each round, final task consists in physically relocating the data items for which new locations have been derived, by triggering the state transfer to move the objects according to the relocation map encoded in the PAAs.

As can be deduced from the previous description, the work is divided among all nodes and communication takes place only at the end of tasks 2, 5, and 6, to exchange statistical information, distribute the PAA, and finally to relocate the objects. Even the tasks that require communication are performed in parallel, without the help of any centralized component.

III. OPTIMIZER

The problem of identifying the best allocation of data to nodes has been already studied in various variants considering different system and cost models[28], [21] (for instance, the cache location problem[17], the file allocation problem[14], just to name a few). Most of these works assume that the objective and constraint functions can be expressed (or approximated) via linear functions, and accordingly formulate an Integer Linear Programming (ILP) problem.

The ILP model can 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 contain). In Table I we list the parameters used in the problem formulation, which of minimizing the following cost function:

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

subject to the following constraints:

$$\forall i \in \mathcal{O} : \sum_{j \in \mathcal{N}} X_{ij} = d \wedge \forall j \in \mathcal{N} : \sum_{i \in \mathcal{O}} X_{ij} \leq C_j \quad (2)$$

Despite its convenient mathematical formulation, the above ILP problem is not scalable in our target scenario, as it would require to collect and gather access statistics of all nodes for all the objects in the system. This would impose prohibitive overheads both in terms of the local storage needed to maintain the access statistics to all objects at each node, as well as in terms of the amount information exchanged among the various nodes. Even assuming that these costs were affordable, the number of decision variables would be proportional to the total number of objects in the system, which would preclude its viability in big-data systems. Finally, ILP is a well-known NP-hard, and hence inherently a non-scalable problem[28]. These considerations motivated us to design an efficient approximation algorithm that relies on three complementary approaches for efficient computation, which we present in a progressive fashion for the sake of clarity.

In AUTOPLACER we tackle these drawbacks by introducing a lightweight, multi-round distributed optimization algorithm, which, at each round, optimizes the placement of the top- k most accessed objects by each node. For the sake of presentation, we will present the AUTOPLACER's distributed optimizer in an incremental fashion. Namely, for clarity of exposition, we start by describing a centralized approach in which a single node gathers the data access frequency information collected, by each node, using independent Space-Saving Top- k algorithm instances.

A. Space-Saving Top- k algorithm

An important building block of AUTOPLACER is the Space-Saving Top- k algorithm by Metwally *et al.* [24]. 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.

Internally, the Space-Saving Top- k algorithm keeps a configurable number m (where $m \ll |\mathcal{O}|$) of counters, which are used to keep track of the frequencies of a subset of the objects accessed in the stream. The Space-Saving Top- k algorithm is extremely simple and lightweight, but, interestingly, its space-complexity can be tuned to bound the maximum error introduced in the frequency tracking. Basically, whenever an object is accessed, it is checked whether this is already monitored by one of the counters. If this is the case, the corresponding counter is increased. Otherwise, if there are still free available counters, the item is assigned to one of them and starts being tracked. When all available counters are assigned, the item with the smallest frequency of access is evicted, and the new item is added with the frequency of the evicted item plus one.

The drawback of using this statistical method is that the Top- k list may not be accurate in terms of both the elements that compose it and their estimated frequency. However, Space-Saving Top- k algorithm ensures that the access frequencies of the objects it tracks are always consistently

overestimated, and that the maximum overestimation error is equal to the frequency of the last element tracked in the Top- k , denoted as F_k . Further, it is possible to show that, independently of the distribution of accesses to items in the stream, it is possible to bound the error, ε_i of the frequency of each item i tracked in Top- k to $\varepsilon_i \leq \epsilon F_k$, where $\epsilon \in (0, 1]$ is a user-defined parameter, by using a number of counters m :

$$m = \frac{1}{\epsilon} * \frac{N}{F_k}$$

where N is the length of the stream.

B. Using Approximate Information

As already mentioned, the optimization algorithm used by AUTOPLACER determines, in each round, the optimal placement for the objects most frequently accessed (read/updated) during that round. These object, and their corresponding access frequencies are identified using the Space-Saving Top- k algorithm recalled in the previous section, of which each node j runs 2 distinct instances (noted as $top-k_j^d$, resp. $top-k_j^{wr}$) to identify the k data items that it most frequently read, resp. written during the current optimization round.

Let us denote with $top-k_j(\mathcal{O})$ the subset of k data items \mathcal{O} contained in the read and write Top- k instances at node j , and with $top-K(\mathcal{O}) = \cup_{j \in \mathcal{N}}(top-k_j(\mathcal{O}))$ the union of the Top- k data items accessed by the entire set of 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, but 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}).

Concerning incompleteness, in order to instantiate the previously illustrated ILP problem for the data items in $top-K(\mathcal{O})$, we need to define what access frequencies to attribute to the data items not present in the Top- k of a node j , but present in the Top- k of another node l . To address this issue, we simply set to 0 the frequencies $\hat{r}_{il}, \hat{w}_{il}$ of an object $i \notin top-k_l(\mathcal{O})$.

Concerning the approximate nature of the information provided by Top- k , we note that this may have an impact on the quality of the identified solution, namely the data assignment matrix \hat{X} identified using the approximate access frequencies provided by Top- k may differ from the matrix X^{Opt} obtained using exact information on data popularity.

An interesting question is therefore how degraded is the quality of the data placement solution when using Top- k . In the following theorem we provide an answer to this question by deriving an upper bound on the approximation ratio of the proposed algorithm. Our proof shows that the approximation ratio is a function of the maximum approximation error provided by any $top-k_j(\mathcal{O})$, 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 $top-K(\mathcal{O})$:

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

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

$$\begin{aligned} rR &= \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top-K(\mathcal{O})} \bar{X}_{ij}^{Opt} cr^r r_{ij}}{O(N - d)} \\ rW &= \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top-K(\mathcal{O})} \bar{X}_{ij}^{Opt} cr^w w_{ij}}{O(N - d)} \\ lR &= \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top-K(\mathcal{O})} X_{ij}^{Opt} cl^r r_{ij}}{Od} \\ lW &= \frac{\sum_{j \in \mathcal{N}} \sum_{i \in top-K(\mathcal{O})} X_{ij}^{Opt} cl^w w_{ij}}{Od} \end{aligned}$$

We can then rewrite Opt and derive its lower bound:

$$\begin{aligned} Opt &= |top-K(\mathcal{O})|(|\mathcal{N}| - d)(cr^r rR + cr^w rW) + \\ &\quad + d(cl^r lR + cl^w lW) \geq \\ &\geq |top-K(\mathcal{O})|(|\mathcal{N}| - d)(cr^r rR + cr^w rW) \end{aligned} \quad (3)$$

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 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:

$$C(\hat{X}, r_{ij}, w_{ij}) \leq Opt + Ode^*(cr^r + cr^w) \quad (4)$$

The approximation ratio is therefore:

$$\frac{C(\hat{X}, r_{ij}, w_{ij})}{C(X^{Opt}, r_{ij}, w_{ij})} \leq 1 + \frac{d}{(N - d)} \frac{e^*(cr^r + cr^w)}{cr^r rR + cr^w rW} \quad (5)$$

In the following corollary we exploit the bounds on the space complexity of the Space-Saving Top- k algorithm [24] to estimate the number of distinct counters to use to achieve a target approximation factor $1 + \frac{d}{N-d}\phi$.

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

$$m = \frac{\text{StreamLength} \cdot cr^r rR + cr^w rW}{\phi \cdot (cr^r + cr^w)}$$

where *StreamLength* 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 [24], 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 $m = \frac{\text{StreamLength}}{\epsilon F_k}$ counters. ■

C. Optimizer Parallelization

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

The ILP problem[28], which requires decision variables to be integers and is computationally onerous. Therefore, we transform it into an efficiently solvable linear programming (LP) problem. To this end, we let the matrix X assume real values between 0 and 1 (adding an extra constraint $\forall i, \forall j, 0 \leq X_{ij} \leq 1$). Note that the solutions of the LP problem can have real values, hence each object is assigned to the d nodes which have highest X_{ij} values. As in [28], 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 $d + i$ -th ($i \in [1, N - d]$) highest scores.

Second, we introduce a controlled relaxation of the capacity constraint 2, which allows us to partition the ILP problem into $|\mathcal{N}|$ independent optimizations problems that we solve in parallel across the node of the platform. To this end, each node groups the access statistics collected by its top- k by the home node of the corresponding key. Let $\text{top-}k(\mathcal{O})_j^i$ be the set of keys in the top- k of node j that have i as home node. Each node j sends its $\text{top-}k(\mathcal{O})_j^i$ to each other node i in the system, and gathers the access statistics $\cup_{j \in \mathcal{N}} \text{top-}k(\mathcal{O})_i^j$ concerning the data of which it is the home node.

At this point each node computes the new placement for the data in $\cup_{j \in \mathcal{N}} \text{top-}k(\mathcal{O})_i^j$, of which it is the home node. Note that since we are instantiating the (I)LP optimization problems in parallel, and in an independent fashion, we need to take an extra countermeasure to guarantee that the capacity constraints are not violated. To this end it is sufficient to instantiate the ILP problems, at each node $j \in \mathcal{N}$ with a capacity $C'_j = C_j - |\mathcal{N}|k$. In practice, this relaxation is expected to have minimum impact on the solution quality as $k \ll C_j$.

D. Turn-based Self-Stabilizing Optimization

The distributed optimization algorithm advances in each rounds, feeding as input for each round, the content of the Top-

k most accessed items during that round by each node in the system. Note that if we assume 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, as typically the case of realistic workloads [22], [28], the Top- k lists would tend to track the very same objects (i.e., the most popular ones) along every round.

In order to circumvent this issue, in each round, AUTOPLACER tracks, in the Top- k lists of each node, only the keys whose placement has not been optimized in previous rounds. This guarantees that, in two different rounds, two disjoint set 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 [15], i.e. the continuous re-location of a key between two or more nodes, as it guarantees that the position each object is optimized/updated at most once. Note that this implies that, if we assume stable data access distributions, the approximation ratio achieved by the round-based optimization algorithm will be able to guarantee the bound introduced in Theorem 1. At each round, in fact, 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.

In order to avoid analyzing the “tail” of the data access distribution, whose optimization would lead to negligible overheads, AUTOPLACER relies on a simple self-stabilizing mechanism that halts the distributed optimization algorithm in case the “gain” achieved during the last optimization round does not exceed a user-tunable minimum threshold. In AUTOPLACER we opted for choosing, as metric to evaluate the gain of an optimization round, the reduction, ΔC , of the cost function C (eq. 1) achieved during the last optimization round. This is achieved by letting each node j computes the difference ΔC_j between the value of the cost function C before and after the re-location of the objects whose placement j has optimizing in the current round. This information, along with the updated placement map for the optimized keys (which, as discussed in the next section, is encoded in a PAA), is disseminated across the nodes of the system, which can then compute $\Delta C = \sum_{j \in \mathcal{N}} \Delta C_j$ and, deterministically, evaluate the predicate on the termination of the optimization algorithm.

IV. PROBABILISTIC ASSOCIATIVE ARRAY

The Probabilistic Associative Array (PAA) is a novel data structure that allows encoding a lookup table, in a space efficient, but approximate way. We present the PAA as an abstract data type, with an interface analogous to conventional associative arrays. Next, we discuss how they can be implemented. We will delegate the discussion on how PAAs are used by AUTOPLACER to efficiently track the placement of re-located objects to Section V.

A. PAA Abstract Data Type

The PAA is abstractly specified by the API reported in Table II. The exposed API is similar to that of a conventional associative array, including methods to create and query a map between keys and (constant a -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

Method	Input Parameters	Output
CREATE	Set(Key, Value[a]), α , β	PAA
GET	Key	Value[a]
ADD	Set(Key, Value[a])	PAA
GETDELTA	PAA	Δ PAA
APPLYDELTA	Δ PAA	PAA

Table II
PAA INTERFACE.

a set of pairs in the domain ($\text{key} \times \text{array}[a]$ of values) to be stored in the PAA (called, succinctly, *seed map*) and two tunable error parameters α and β (to be discussed shortly); 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 set of pairs in the domain ($\text{key} \times \text{array}[a]$ of values), and store them in an existing PAA.

As already mentioned, the PAA tradeoffs accuracy for space efficiency, and may return erroneous 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., to provide a non-NULL return value 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 NULL for a key contained in the seed map.
- it may return an *incorrect* array of values for a key contained in the seed map. The probability of returning inaccurate arrays is controlled by parameter β .
- 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 Δ PAA, of the differences between the base PAA over which the method is invoked (say PAA₁) with respect to the input PAA, say PAA₂. The Δ PAA returned by GETDELTA can then be used to obtain PAA₂ by invoking the method APPLYDELTA over PAA₁ and passing as input parameter Δ PAA.

B. FeatureExtractor Key Interface

As already mentioned in the introduction, the PAA relies on machine-learning techniques to encode, in a space efficient way, the mapping between keys and values. 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 $\langle \text{featureName name, featureValue value} \rangle$, where *featureName* is a set of n distinct strings, defining the “feature space” of the application [25], and *featureValue* is a continuous (real or integer) 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 n -dimensional feature-space that can be more efficiently analyzed and partitioned by a statistical inference tool. Features can be “naturally” derived from the data model used in the application. For instance, if an object-oriented (or relational model) is used, the key corresponding to an object of class C with $\text{ID}=id$ could be associated with a feature encoded by the pair $\langle C, id \rangle$. This can be illustrated in a more concrete way considering the particular case of the TPC-C benchmark used in our evaluation. In this case, a “Customer” object c would be associated with a feature, $\langle \text{“Customer”, } c \rangle$. Further, since in TPC-C a customer is statically registered in a “Warehouse” object, c would have a second feature $\langle \text{“Warehouse”, } w \rangle$, being w the identifier of the warehouse where c is registered.

Note that this sort of feature extraction can be easily automated, provided the availability 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 [3] or domain specific languages [5].

C. PAA Implementation

Internally, a PAA is composed by 2 modules: a *Scalable Bloom filter* (SBF) [1] and an set of *a rule sets*, generated by a independent instances of VFDT, i.e. an incremental decision tree algorithm designed to operate over a data stream [13]. The purpose of the SBF is to determine if a key has been inserted in the PAA, while the purpose of the rule sets is to store the values associated with a key in the PAA. Before discussing how these building blocks are used to build a PAA, we briefly overview them.

Scalable Bloom filters (SBF) are a variant of Bloom filters (BF) [4], a well know data structure that supports probabilistic test for membership of elements in sets. A BF never yield 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. SBFs achieve adaptation by instantiating, on demand, a sequence of BFs with increasing capacity.

VFDT 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 [25]. Conversely, 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.

We can now discuss how the PAA API reported in Table II is implemented:

- **CREATE**: a SBF is created, sizing it to ensure the target error rate α and populating it with the keys passed as input parameter. Further we train a new instances of VFDT, one for each entry in the Value array passed as input parameter. The i -th instance of VFDT ($i \in [1, \dots, a]$) 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. This allows us to tightly control is 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 .

- **GET**: queries for a key k are performed by first querying the SBF. In the response is negative, null 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, which may return any of the target classes that it observed during the training phase.

- **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 contained in the seed map 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 seed map that is being added to the PAA. In this phase we control the learning of the new mapping in a single batch, by allowing the VFDT algorithm to scan the new training set multiple times until we the target bound on misclassification β is satisfied.

- **GETDELTA**: the output consists of the the binary diff of the SDFs, plus the ruleset of the target PAA.

- **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 ruleset of the output PAA is simply set equal to the one contained in the input ΔPAA .

V. AUTOPLACER: GLOBAL ALGORITHM AND EXTENSIONS

We can now provide, in Algorithm 1 and Algorithm 2, the pseudo-code formalizing the behavior of the AUTOPLACER optimization algorithm executed by a node j . Each node maintains an array, denoted as *curPAA*, that maintains one PAAs per each node j in the system. Specifically, the j -th entry of *currPAA* contains the PAA that is used to keep track of the objects that were, originally assigned to node j by the consistent hashing, and then re-located to some other node by AUTOPLACER during an optimization round.

At each round, n collects for a period of time statistics concerning its Top- k most frequently read/updated data items. As discussed in Section III-D, to avoid stagnation of the Top- k lists, we track only the accesses to items whose placement

```

1 Array[1...|N'|] of PAA : curPAA={⊥, ..., ⊥}
2 PAA: newPAA=⊥
3 do
4   Array[1...|N'|] of Set<i,  $\hat{r}_{ij}$ ,  $\hat{w}_{ij}$ > : reql=⊥, reqr=⊥
5   <topkd, topkw> ← collectStats(curPAA)
6   foreach node j ∈ Π do
7     foreach <i,  $\hat{r}_{ij}$ ,  $\hat{w}_{ij}$ > ∈ {topkd ∪ topkw} do
8       if hash(i) == j then
9         reql[j] ← reql[j] ∪ <i,  $\hat{r}_{ij}$ ,  $\hat{w}_{ij}$ >
10      end
11    end
12  end
13  foreach node j ∈ Π do
14    send(reql[j]) to j
15    reqr[j] ← receive() from j
16  end
17  < $\hat{X}$ ,  $\Delta_{C^*}$ > ← solveLP(reqr)
18  newPAA ← curPAA[n].ADD( $\hat{X}$ )
19   $\Delta PAA$ : delta ← newPAA.GETDELTA(curPAA[n])
20  broadcast(delta)
21  foreach node j ∈ Π do
22    delta ← receive() from j
23    curPAA[j] ← curPAA[j].APPLYDELTA(delta)
24  end
25  moveData()
26 while  $\Delta_{C^*} > \alpha$ ;

```

Algorithm 1: AUTOPLACER's behavior at node n

had not been previously optimized. To identify the set of keys whose placement was already optimized without maintaining an explicit directory (which may grow unbounded and constrain scalability), we rely, again on PAAs, providing them as input parameter to the primitive that is used in the pseudo-code to encapsulate the usage of the Space-Saving Top- k algorithm, namely *collectStatics*.

Next, n groups the access frequencies for the objects i in its Top- k lists based on their home nodes according to consistent hashing, and sends them this information (see line 6-14). In parallel, node n waits to gather the access statistics from the other nodes concerning the objects for which n is a home node, again according to consistent hashin (line 15).

At this point n solves the linear programming optimization problem for the objects he is responsible of during this optimization round (primitive *solveLP*, line 17), determining their new data placement (encoded in the matrix \hat{X}) and the reduction in the cost function (denoted as Δ_{C^*}). Next, see lines 18-20, he updates its own PAA, computes the delta with respect to the previous iteration and disseminates it to the other nodes. Following, he gathers the updated PAAs from the other nodes (lines 21-24), and triggers the re-location of the data via the *moveData()* primitive. This primitive will use the updated PAAs to determine the set of items which have been re-located in this round from node n to some other node, and transfer them to the intended recipient.

Algorithm 2 shows the pseudocode for the lookup function for a key k . First, consistent hashing is used to identify the original home node of k , say n' . We then check whether the PAA associated with n' contains k . In the positive case, we use the mapping information provided by *currPAA*[n'] 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 the consistent hashing.

The PAA supports the lookup function in AUTOPLACER, by providing information on the new owners of relocated items.


```

1 Array[1...d] of Nodes LOOKUP(Key k)
2   if currPAA[hash(k)].GET(k)  $\neq$   $\perp$  then
3     return currPAA[hash(k)].GET(k)
4   else
5     return hash(k)
6   end
7 end

```

Algorithm 2: PAA-based lookup function

If no mapping is found in the PAA, AUTOPLACER exploits the default consistent hashing to locate the item. In the following is based on probabilistic mechanisms that explore semantic information that may be encoded in the object keys, with the intention of reducing the communication overhead and local storage required when compared with the original full relocation map.

A. Dealing with join/leaves and dynamic workloads

We have presented so far the AUTOPLACER assuming a static group of nodes and stable workloads. As AUTOPLACER is designed to use consistent hashing for the vast majority of seldom accessed data items, the addition and removal of nodes does not generate substantial problems. In this sense, AUTOPLACER inherits, from consistent hashing, the desirable property of minimizing the number of items relocated in the system upon the join or leave of a node from the system. Concerning the objects re-located by AUTOPLACER to nodes that leave the system, these can be simply relocated, upon detection of the group membership change to the original home node according to consistent hashing.

Finally, in presence of constantly fluctuating workloads the effectiveness of AUTOPLACER (and arguably of any self-tuning system) is clearly challenged. However, one may extend AUTOPLACER with simple mechanisms to detect abrupt changes of the data access distributions, e.g. by tracking significant variations of the probability of remote data accesses [18], and trigger a new round of optimization.

VI. 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. Finally, we show the impact on the system performance of the AUTOPLACER placement optimization strategy.

A. Experimental Setting

All results presented in this section were obtained by running a prototype of AUTOPLACER based on Infinispan and a non-modified version of Infinispan as baseline. As benchmarking hardware, we have used 40 virtual machines, running on Xen, where each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet. For our experimental study we use the TPC-C benchmark [22]. TPC-C is a well-known benchmarks, representative of OLTP environments and characterized by complex and heterogeneous transactions, with very skewed access patterns and high conflict probability.

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

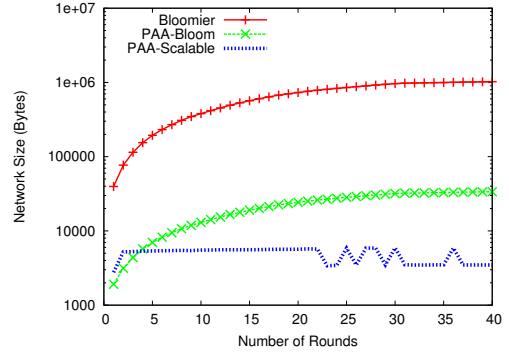


Figure 1. Network data exchanged between nodes when using different associative arrays

locality patterns in the data accesses of each node. This modification consists in configuring the benchmark with a number of warehouses equal to the number of nodes (40), and creating a probability of the transactions from each node to access the warehouse with the same index as the node. This simulates a load balancer which with some controllable probability dispatches the transactions perfectly. So, for example, in the 90% locality scenario, nodes will have disjoint data access patterns (each accessing a different warehouse) for 90% of the transactions, while the remaining 10% access data uniformly as in an unmodified TPC-C. Notice however, that the data is still scattered randomly by the nodes according to consistent hashing.

B. Probabilistic Associative Array

In this section we study tradeoffs in the the space efficiency and accuracy involved in the configuration and implementation of the PAA. For these results, we have configured the benchmark with 100% locality. In order to use the PAA with TPC-C, we have modified the TPC-C keys such that they export the Feature Extractor Interface according to the static attributes of the objects they represent.

a) *Bloom Filter*: Figure 1 presents the network bandwidth of different implementations of the PAA, compared with another form of probabilistic associative arrays, the Bloomier Filters (BLOOMIER) [8] as the rounds advance in the system. One implementation of the PAA uses regular Bloom filters (PAA-BLOOM) [4], while the other uses a scalable Bloom filter (PAA-SCALABLE) [1]. Both PAAs were configured with $\alpha = 0.01$ and the Bloomier filter's false positive probability was set to 0.01.

It can be observed 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. Table III shows the error α associated with each implementation and the correspondent local storage requirements at the end of the experiment. As it can be seen, PAA-SCALABLE has very low error probabilities, which come at the cost of increased local size. However, its space requirements are still considerably smaller than those of the Bloomier filter.

b) *Machine Learner*: Figure 2 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

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

Table III
RE-LOCATED OBJECTS AND SIZE OF DIFFERENT PAA IMPLEMENTATIONS

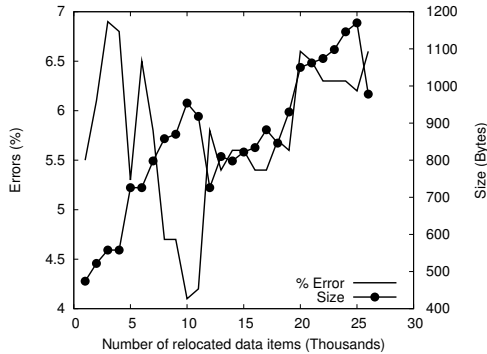


Figure 2. Error probability and rules size for VFDT

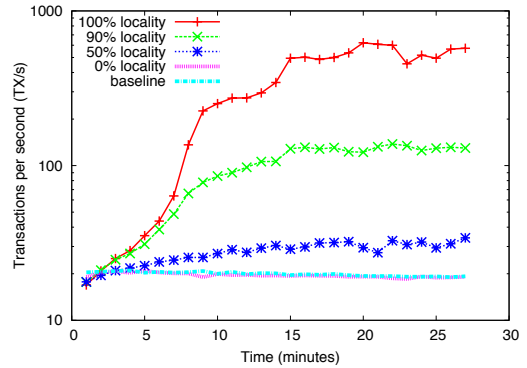
moved in the system, the number of rules increases, leading to an increase in the size taken by this portion of the PAA. However, the machine learner can represent the mapping of 26600 keys in 1000 Bytes, which correspond to 213 rules. Furthermore, it can also be observed that while a significant number of keys is added to the machine learner (around 1000 per round), the error remains relatively stable.

C. Leveraging from Locality

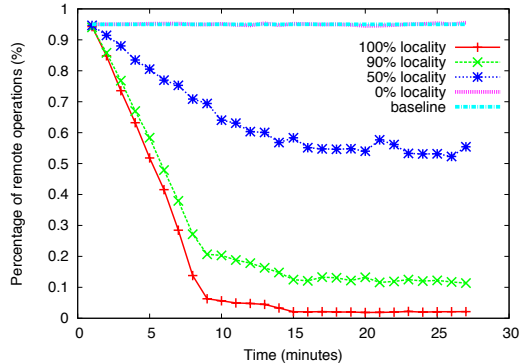
This section shows how AUTOPLACER is able to leverage from locality patterns in the workload. The results were obtained with TPC-C, adapted as explained before, in a read/write scenario with 50% of writes, and with a replication of degree $d = 2$.

Figure 3(a) shows the throughput of AUTOPLACER, compared with the non optimized system 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. As times passes, and more rounds of optimization take place, the system throughput increases up to a point where no further optimization is performed. In is interesting to note that, in the 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).

Figure 3(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



(a) Throughput with varying degrees of locality.



(b) % of remote operations.

Figure 3. AUTOPLACER performance

optimized system, the average the 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 decrease in time in a system optimized by AUTOPLACER. The plots also show another interesting aspect of the system operation. 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 touch a large number of objects, thus multiple rounds of optimization are required to alleviate the network, which is the bottleneck in this heavy load setting. This clearly highlights the relevance 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.

VII. RELATED WORK

A common approach to implementing data placement mechanisms in large scale systems is to manage the data through coarse grain by partitioning it into buckets (also named directories [10] or tablets [9]). 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 partitioning allows for somewhat manageable index sizes, the effectiveness

of the load balancing is limited. 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) [10], [9], [19]. AUTOPLACER does not require the programmer to manually bucket items. While we benefit from information enabled in the key structure, this information is not used for object placement, it is only used for optimizing the PAA. Also our system can establish a fine grain placement for the most accessed items.

As hinted several times in the paper, there is extensive work of finding optimal data placement strategies in multiple contexts. Many of these systems, such as Ursa [28] or Schism [11], attempt to perform optimization at a finer grain than buckets, but require the use centralized components to compute the placement and to keep the resulting directories with the relocation map. As a result, they suffer from scalability limitations as the number of data items grows.

Several works have also attempted to derive distributed versions of the placement algorithm, to avoid the bottleneck of a single centralized component. The work by Leff *et al* proposes several distributed algorithms to approach the replica placement problem [21], and improvements to this work have been recently proposed in [20] and [29]. These results are not applicable to our system, as they only consider the placement of read-only replicas and not of the object ownership. Furthermore, this solution attempts to relocate all the data in the system, which may lead to scalability limitations similar to those of Ursa or Schism.

The work by Vilaça *et al.* [27] presents a Space-Filling Curves-based [26] 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 thus has no relation with the actual data access patterns that may emerge when running the system.

VIII. CONCLUSIONS

We have presented AUTOPLACER, a system aimed at self-tuning the 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 1-hop routing latency using a novel probabilistic data structure, the PAA, that minimize the cost of maintaining and disseminating the data relocation map. AUTOPLACER has been implemented as a variant of the JBoss Infinispan key-value store and the resulting prototype experimentally evaluated. The results shown that AUTOPLACER can achieve a throughput 50 times better than the original Infinispan implementation based on consistent hashing.

REFERENCES

- [1] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007.
- [2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proc. of the 4th USITS*, Seattle (WA), USA, 2003.
- [3] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.
- [4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.
- [5] J. Cachopo and A. Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *ICWE*, pages 297–304, 2006.
- [6] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. of the 7th OSDI*, pages 15–15, Seattle (WA), USA, 2006.
- [7] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*. Springer, 2010.
- [8] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. of the 15th SODA*, pages 30–39, New Orleans (LA), USA, 2004.
- [9] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!’s hosted data serving platform. In *Proc. of the 34th VLDB*, volume 1, Auckland, New Zealand, August 2008.
- [10] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally-distributed database. In *Proc. of the 10th OSDI*, pages 251–264, Hollywood, CA, USA, 2012.
- [11] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *Proc. of the 36th VLDB*, Singapore, September 2010.
- [12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. In *Proc. of the 21st SOSP*, pages 205–220, Stevenson (WA), USA, 2007.
- [13] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the 6th KDD*, pages 71–80, Boston, MA, USA, 2000.
- [14] L. Dowdy and D. Foster. Comparative models of the file assignment problem. *ACM Comput. Surv.*, 14(2):287–313, 1982.
- [15] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. *SIGOPS Oper. Syst. Rev.*, 23(5):211–223, November 1989.
- [16] S. Garbatov and J. Cachopo. Data access pattern analysis and prediction for object-oriented applications. *INFOCOMP Journal of Computer Science*, 10:1–14, december 2011.
- [17] P. Krishnan, Danny Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, October 2000.
- [18] Sangyeol L. and Taewook L. Cusum test for parameter change based on the maximum likelihood estimator. *Sequential Analysis: Design Methods and Applications*, 23(2):239–256, 2004.
- [19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [20] N. Laoutaris, O. Telelis, V. Zissimopoulos, and I. Stavrakakis. Distributed selfish replication. *IEEE Trans. Parallel Distrib. Syst.*, 17(12):1401–1413, December 2006.
- [21] A. Leff, J. Wolf, and P. Yu. Replication algorithms in a remote caching architecture. *IEEE Trans. Parallel Distrib. Syst.*, 4(11):1185–1204, November 1993.
- [22] S. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. In *Proc. of the SIGMOD Conf.*, pages 22–31, Washington, D.C., United States, 1993.
- [23] F. Marchioni and M. Surtani. *Infinispan Data Grid Platform*. PACKT Publishing, 2012.
- [24] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th ICDT*, pages 398–412, Edinburgh, Scotland, 2005.
- [25] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [26] H. Sagan and J. Holbrook. *Space-Filling Curves*. Springer-Verlag New York, 1994.
- [27] R. Vilaça, R. Oliveira, and J. Pereira. A correlation-aware data placement strategy for key-value stores. In *Proc. of the 11th DAIS*, pages 214–227, Reykjavik, Iceland, 2011.
- [28] G.-Won You, S.-Won Hwang, and N. Jain. Scalable load balancing in cluster storage systems. In *Middleware*, pages 101–122, Lisbon, Portugal, 2011.
- [29] S. Zaman and D. Grosu. A distributed algorithm for the replica placement problem. *IEEE Trans. Parallel Distrib. Syst.*, 22(9):1455–1468, September 2011.