View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

A correlation-aware data placement strategy for key-value stores *

Ricardo Vilaça, Rui Oliveira, and José Pereira

High-Assurance Software Laboratory
University of Minho
Braga, Portugal
{rmvilaca,rco,jop}@di.uminho.pt

Abstract. Key-value stores hold the unprecedented bulk of the data produced by applications such as social networks. Their scalability and availability requirements often outweigh sacrificing richer data and processing models, and even elementary data consistency. Moreover, existing key-value stores have only random or order based placement strategies.
In this paper we exploit arbitrary data relations easily expressed by the application to foster data locality and improve the performance of complex queries common in social network read-intensive workloads.
We present a novel data placement strategy, supporting dynamic tags, based on multidimensional locality-preserving mappings.
We compare our data placement strategy with the ones used in existing key-value stores under the workload of a typical social network application and show that the proposed correlation-aware data placement strategy offers a major improvement on the system's overall response time and network requirements.

Keywords: Peer-to-Peer; DHT; Cloud Computing; Dependability

1 Introduction

Highly distributed and elastic key-value stores are at the core of the management of sheer volumes of data handled by very large scale Internet services. Major examples such as Google, Facebook and Twitter rely on key-value stores to handle the bulk of their data where traditional relational database management systems fail to scale or become economically unacceptable. To this end, distributed key-value stores invariably offer very weak consistency guarantees and eschew transactional guarantees. These first generation distributed key-value stores are built by major Internet players, like Google [4], Amazon [8], FaceBook [17] and Yahoo [6], by embracing the Cloud Computing model.

While common applications leverage, or even depend on, general multi-item operations that read or write whole sets of items, current key-value stores only

* Partially funded by the Portuguese Science Foundation (FCT) under project Stratus – A Layered Approach to Data Management in the Cloud (PTDC/EIA-CCO/115570/2009) and grant SFRH/BD/38529/2007.

offer simple single item operation or at most range queries based on the primary key of the items [23]. These systems require that more general and complex multi-item queries are done outside of the system using some implementation of the Map Reduce[7] programming model: Yahoo's PigLatin, Google's Sawzall, Microsoft's LINQ.

However, if the API does not provide enough operations to efficiently retrieve multiple items for the general multi-item queries they will have a high cost in performance. These queries will mostly access a set of correlated items. Zhonk et al. have shown that the probability of a pair of items being requested together in a query is not uniform but often highly skewed [27]. They have also shown that correlation is mostly stable over time for real applications. Furthermore, when involving multiple items in a request to a distributed key-value store, it is desirable to restrict the number of nodes who actually participate in the request. It is therefore more beneficial to couple related items tightly, and unrelated items loosely, so that the most common items to be queried by a request would be those that are closed to each other.

Leveraging the items correlation and the biased access patterns requires the ability to reflect that correlation into the items placement strategies[26]. However, data placement strategies in existing key-value stores [4,6,8,17] only support single item or range queries. If the data placement strategy places correlated items on the same node the communication overhead for multi-items operations is reduced. The challenge here is to achieve such placement in a decentralized fashion, without resorting to a global directory, while at the same time ensuring that the storage and query load on each node remains balanced.

We address this challenge with a novel correlation-aware data placement strategy that allows the use of dynamic and arbitrary tags on data items and combines the usage of a Space Filling Curve (SFC) with random partitioning to store and retrieve correlated items. This strategy was built into DataDroplets, an ongoing effort to build an elastic data store supporting conflict-free strongly consistent data storage. Multi-item operations leverage disclosed data relations to manipulate dynamic sets of comparable or arbitrarily related elements. DataDroplets extends the data model of existing key-value stores with tags allowing applications to establish arbitrary relations among items. It is suitable to handle multi-tenant data and is meant to be run in a Cloud Computing environment.

DataDroplets supports also the usual random and order based strategies. This allows it to adapt and to be optimized to the different workloads and, in the specific context of Cloud Computing, to suit the multi-tenant architecture. Moreover, as some data placement strategies may be non-uniform, with impact on the overall system performance and fault tolerance, we implemented a load balancing mechanism to enforce uniformity of data distribution among nodes.

We have evaluated our proposal with a realistic environment and workload that mimics the Twitter social network.

The remainder of the paper is organized as follows. Section 2 presents DataDroplets and Section 3 describes the correlation-aware placement strategy. Sec-

tion 4 presents a thorough evaluation of the three placement strategies. Section 5 discusses related work and Section 6 concludes the paper.

2 DataDroplets Key-value store

DataDroplets is a key-value store targeted at supporting very large volumes of data leveraging the individual processing and storage capabilities of a large number of well connected computers. It offers a low level storage service with a simple application interface providing the atomic manipulation of key-value items and the flexible establishment of arbitrary relations among items.

In [23] we introduced DataDroplets and presented a detailed comparison against existing systems regarding their data model, architecture and trade-offs.

2.1 Data Modeling

DataDroplets assumes a very simple data model. Data is organized into disjoint *collections* of items identified by a string. Each item is a triple consisting of a unique key drawn from a partially ordered set, a value that is opaque to DataDroplets and a set of free form string tags.

DataDroplets use tags to allow applications to dynamically establish arbitrary relations among items. The major advantage of tags is that they are free form strings and thus applications may use them in different manners.

Applications migrated from relational databases with relationships between rows of different tables and frequent queries over this relationships may have tags as foreign keys. Therefore, they will efficiently retrieve correlated rows.

Social applications may use as tags the user's ID and the IDs of the user's social connections allowing that most operations will be restricted to a small set of nodes. Also, tags can be used to correlate messages of the same topic.

2.2 Overlay management

DataDroplets builds on the Chord [22] structured overlay network. Physical nodes are kept organized on a logical ring overlay where each node maintains complete information about the overlay membership as in [12,18]. This fits our informal assumptions about the size and dynamics of the target environments (tens to hundreds of nodes with a reasonably stable membership) and allows efficient one-hoping routing of requests [12].

On membership changes (due to nodes that join or leave the overlay) the system adapts to its new composition updating the routing information at each node and readjusting the data stored at each node according to the redistribution of the mapping interval. In DataDroplets this procedure follows closely the one described in [12].[1]

[1] To the reviewer: since in this paper we do not assess the impact of dynamic membership changes and because the algorithm has been described in [12], we omit most of the details of the procedure.

Besides the automatic load redistribution on membership changes, because some workloads may impair the uniform data distribution even with a random data placement strategy the system implements dynamic load-balancing as proposed in [15]. Roughly, the algorithm is as follows. Periodically, a randomly chosen node contacts its successor in the ring to carry a pairwise adjustment of load.

DataDroplets uses synchronous replication to provide fault-tolerance and automatic fail-over on node crashes [23].

2.3 Data placement strategies

Nodes in the DataDroplets overlay have unique identifiers uniformly picked from the $[0, 1]$ interval and ordered along the ring. Each node is responsible for the storage of buckets of a distributed hash table (DHT) also mapped into the same $[0, 1]$ interval. The data placement strategy is defined on a collection basis.

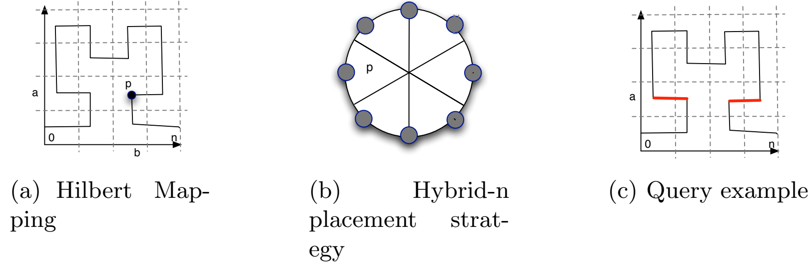
In the following we describe the commonly used data placement strategies for DataDroplets.

The first is the *random placement*, the basic load-balancing strategy present in most DHTs [22,12] and also in most key-value stores [8,6,17]. This strategy is based on a consistent hash function [14]. When using consistent hashing each item has a numerical ID (between 0 and MAXID) obtained, for example, by hashing the item's key. The output of the hash function is treated as a circular space in which the largest value wraps around the smallest value. This is particularly interesting when made to overlap the overlay ring. Furthermore, it guarantees that the addition or removal of a bucket (the corresponding node) incurs in a small change in the mapping of keys to buckets.

The other is the *ordered placement* that takes into account order relationships among items' primary key favoring the response to *range* oriented reads and is present in some key-value stores [6,4,17]. This order needs to be disclosed by the application and can be per application, per workload or even per request. We use an order-preserving hash function [11] to generate the identifiers. Compared to a standard hash function, for a given ordering relation among the items, an order-preserving hash function *hashorder()* has the extra guarantee that if $o1 < o2$, then $hashorder(o1) < hashorder(o2)$.

The major drawback of the random placement is that items that are commonly accessed by the same operation may be distributed across multiple nodes. A single operation may need to retrieve items from many different nodes leading to a performance penalty.

Regarding the ordered placement, in order to make the order-preserving hash function uniform as well we need some knowledge on the distribution of the item's keys [11]. For a uniform and efficient distribution we need to know the domain of the item's key, the minimum and maximum values. This yields a tradeoff between uniformity and reconfiguration. While a pessimistic prediction of the domain will avoid further reconfiguration it may break the uniformity. In the current implementation of DataDroplets the hash function is not made uniform but, as described later, we use a more general approach to balance load.



(a) Hilbert Mapping (b) Hybrid-n placement strategy (c) Query example

Fig. 1. Tagged placement strategy

3 Correlation-aware strategy

A key aspect of DataDroplets is the multi-item access that enables the efficient storage and retrieval of large sets of related data at once. Multi-item operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. The performance of multi-item operations depends heavily on the way correlated data is physically distributed.

The balanced placement of data is particularly challenging in the presence of dynamic and multi-dimensional relations. This aspect is the main contribution of the current work describing a novel data placement strategy based on multi-dimensional locality-preserving mappings. Correlation is derived from disclosed tags dynamically attached to items.

3.1 Tagged placement

The placement strategy, called hereafter *tagged*, realizes the data distribution according to the set of tags defined per item. A relevant aspect of our approach is that these sets can be dynamic. This allows us to efficiently retrieve correlated items, that were previously attached by the application. The strategy uses a dimension reducing and locality-preserving indexing scheme that effectively maps the multidimensional information space to the identifier space, $[0, 1]$.

Tags are free-form strings and form a multidimensional space where tags are the coordinates and the data items are points in the space. Two data items are collocated if they have equal sized set of tags and tags lexicographically close, or if one set is a sub-set of the other.

This mapping is derived from a locality-preserving mapping called Space Filling Curves (SFCs) [19]. A SFC is a continuous mapping from a d-dimensional space to a unidimensional space ($f : N^d \rightarrow N$). The d-dimensional space is viewed as a d-dimensional cube partitioned into sub-cubes, which is mapped onto a line such that the line passes once through each point (sub-cube) in the volume of the cube, entering and exiting the cube only once. Using this mapping,

a point in the cube can be described by its spatial coordinates, or by the length along the line, measured from one of its ends.

SFCs are used to generate the one-dimensional index space from the multidimensional tag space. Applying the Hilbert mapping [3] to this multidimensional space, each data element can be mapped to a point on the SFC. Figure 1(a) shows the mapping for the set of tags $\{a, b\}$. Any range query or query composed of tags can be mapped into a set of regions in the tag space and corresponding to line segments in the resulting one-dimensional Hilbert curve. These line segments are then mapped to the proper nodes. An example for querying tag $\{a\}$ is shown in Figure 1(c), which is mapped into two line segments.

An update to a previous item without knowing its previous tags must find which node has the requested item and then update it. If the update also updates its tags the item will be moved from the old node, defined by old tags, to the new node, defined by new tags.

As this strategy only takes into account tags, all items with the same set of tags will have the same position in the identifier space and therefore will be allocated to the same node. To prevent this we adopt a hybrid-n strategy. Basically, we divide the set of nodes into n partitions and the item's tags instead of defining the complete identifier into the identifier space define only the partition, as shown in Figure 1(b). The position inside the partition is defined by a random strategy. Therefore, the locality is only preserved inter partition.

3.2 Request handling

The system supports common single item operations such as put, get and delete, multi item put (*multiPut*) and get (*multiGet*) operations, and set operations to retrieve ranges (*getByRange*) and equally tagged items (*getByTags*). The details of DataDroplets operations are presented in [23].

Any node in the overlay can handle client requests. When handling a request the node may need to split the request, contact a set of nodes, and compose the clients reply from the replies it gets from the contacted nodes. This is particularly so with multi item and set operations. When the collection's placement is done by tags, this also happens for single item operations.

Indeed, most request processing is tightly dependent of the collection's placement strategy. For the `put` and `multiPut` this is obvious as the target nodes result from the chosen placement strategy.

For operations that explicitly identify the item by key, `get`, `multiGet` and `delete` the node responsible for the data can be directly identified when the collection is distributed at random or ordered. Having the data distributed by tags all nodes need to be searched for the requested key.

For `getByRange` and `getByTags` requests the right set of nodes can be directly identified if the collection is distributed with the ordered and tagged strategies, respectively. Otherwise, all nodes need to be contacted and need to process the request.

4 Experimental evaluation

We ran a series of experiments to evaluate the performance of the system, in particular the suitability of the different data placement strategies, under a workload representative of applications currently exploiting the scalability of emerging key-value stores. In the following we present performance results for the three data placement strategies previously described both in simulated and real settings.

4.1 Test workload

For the evaluation of DataDroplets we have defined a workload that mimics the usage of the Twitter social network.

Twitter is an online social network application offering a simple micro-blogging service consisting of small user posts, the *tweets*. A user gets access to other user tweets by explicitly stating a *follow* relationship, building a social graph.

Our workload definition has been shaped by the results of recent studies on Twitter [13,16,2] and biased towards a read intensive workload based on discussions that took place during Twitter's Chirp conference (the Twitter official developers conference). In particular, we consider just the subset of the seven most used operations from the Twitter API[2] (Search and REST API as of March 2010): `statuses_user_timeline`, `statuses_friends_timeline`, `statuses_mentions`, `search_contains_hashtag`, `statuses_update`, `friendships_create` and `friendships_destroy`.

Twitter's network belongs to a class of scale-free networks and exhibit a small world phenomenon [13]. The generation of tweets, both for the initialization phase and for the workload, follows observations over Twitter traces [16,2]. First, the number of tweets per user is proportional to the user's followers [16]. From all tweets, 36% mention some user and 5% refer to a topic [2]. Mentions in tweets are created by randomly choosing a user from the set of friends. Topics are chosen using a power-law distribution [13].

Each run of the workload consists of a specified number of operations. The next operation is randomly chosen and, after it had finished, the system waits some pre configured time, think-time, and only afterwards sends the next operation. The probabilities of occurrence of each operation and a more detailed description of the workload can be found in [24].

The defined workload may be used with both key-value stores and relational databases.[3]

4.2 Experimental Setting

We evaluate our implementation of DataDroplets using the ProtoPeer toolkit [9]. ProtoPeer is a toolkit for rapid distributed systems prototyping that allows

[2] `http://apiwiki.twitter.com/Twitter-API-Documentation`

[3] The workload is available at `https://github.com/rmpvilaca/UBlog-Benchmark`

switching between event-driven simulation and live network deployment without changing the application code.

For all experiments presented next the performance metric has been the average request latency as perceived by the clients. The system was populated with 64 topics for tweets and a initial tweet factor of 1000. A initial tweet factor of n means that a user with f followers will have $n \times f$ initial tweets. For each run 500000 operations were executed. Different request loads have been achieved by varying the clients think-time between operations. Throughout the experiments no failures were injected.

Simulated setting From ProtoPeer we have used the network simulation model and extended it with simulation models for CPU as per [25]. The network model was configured to simulate a LAN with latency uniformly distributed between 1 ms and 2 ms. For the CPU simulation we have used a hybrid simulation approach as described in [21]. All data has been stored in memory, persistent storage was not considered. Briefly, the execution of an event is timed with a profiling timer and the result is used to mark the simulated CPU busy during the corresponding period, thus preventing other event to be attributed simultaneously to the same CPU. A simulation event is then scheduled with the execution delay to free the CPU. Further pending events are then considered. Therefore, only the network latency is simulated and the other execution times are profiled from real execution. Each node was configured and calibrated to simulate one dual-core AMD Opteron processor running at 2.53GHz.

The system was populated with 10000 concurrent users and the same number of concurrent users were simulated (uniformly distributed by the number of configured nodes).

Real setting We used a machine with 24 AMD Opteron Processor cores running at 2.1GHz, 128GB of RAM and a dedicated SATA hard disk.

We ran 20 instances of Java Virtual Machine (1.6.0) running ProtoPeer. ProtoPeer uses Apache MINA[4] for communication in real settings. We have used Apache Mina 1.1.3. All data has been stored persistently using Berkeley DB Java edition 4.0 [5].

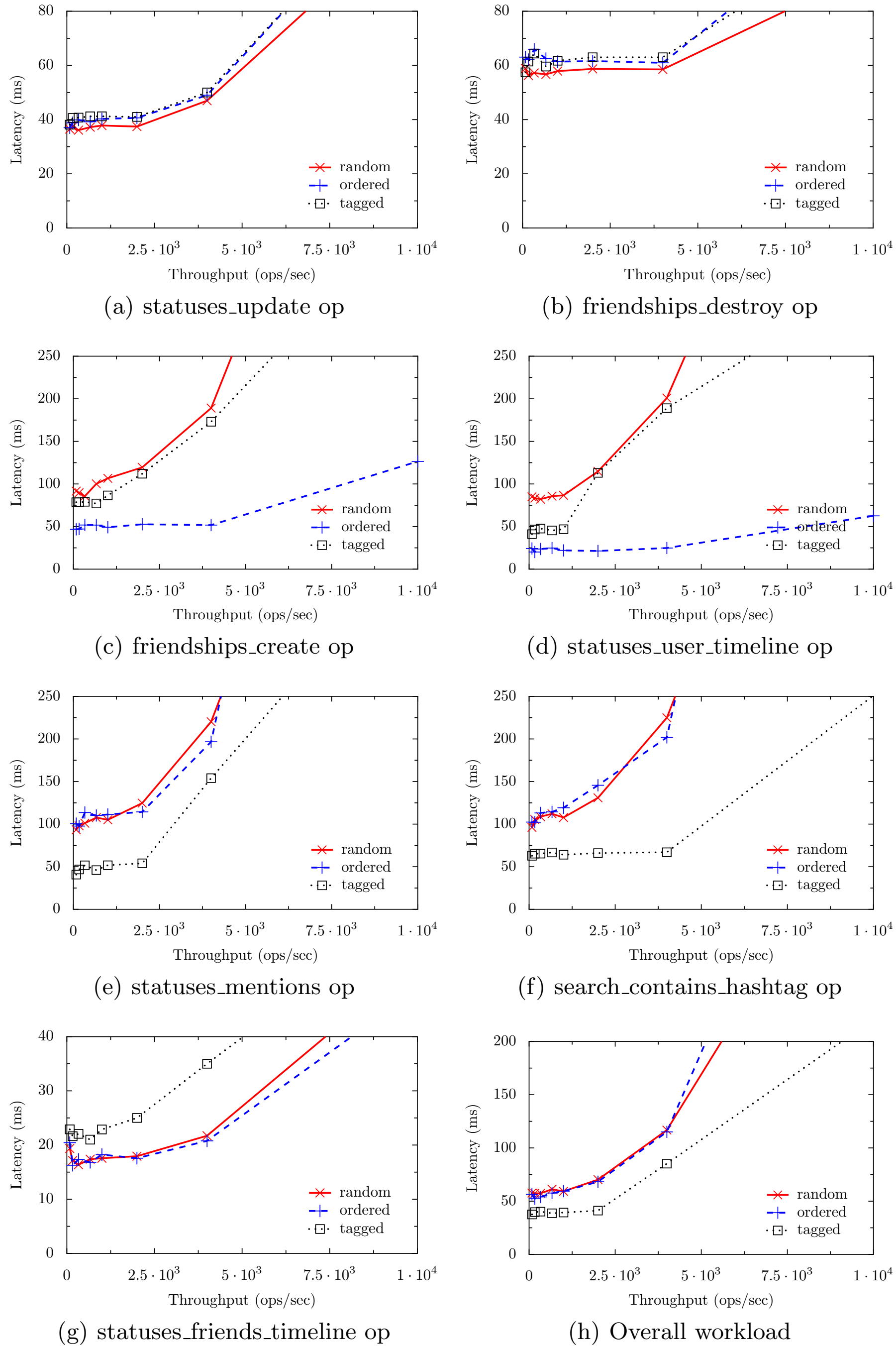
The system was populated with 2500 concurrent users and the same number of concurrent users were run (uniformly distributed by the number of configured instances). During all the experiment IO was not the bottleneck.

4.3 Evaluation of data placement strategies

Simulated setting The graphs in Figure 2 depict the performance of the system when using the different placement strategies available in the simulated setting.

[4] `http://mina.apache.org/`

[5] `http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html`



(a) statuses_update op
(b) friendships_destroy op
(c) friendships_create op
(d) statuses_user_timeline op
(e) statuses_mentions op
(f) search_contains_hashtag op
(g) statuses_friends_timeline op
(h) Overall workload

Fig. 2. System's response time with 100 simulated nodes

The workload has been firstly configured to only use the random strategy (the most common in existing key-value stores), then configured to use the ordered placement for both the `tweets` and `timelines` collections (for `users` placement has been kept at random), and finally configured to exploit the tagged placement for `tweets` (`timelines` were kept ordered and `users` at random). The lines random, ordered and tagged in Figure 2 match these configurations.

We present the measurements for each of the seven workload operations (Figure 2(a) through 2(g)) and for the overall workload (Figure 2(h)). All runs were carried with 100 nodes.

We can start by seeing that for write operations (`statuses_update` and `friendships_destroy`) the system's response time is very similar for all scenarios (Figures 2(a)and 2(b)). Both operations read *one* user record and subsequently add or update one of the tables. The costs of these operations is basically the same in all the placement strategies.

The third writing operation, `friendships_create`, has a different impact, though (Figure 2(c)). This operation also has a strong read component. When creating a follow relationship the operation performs a `statuses_user_timeline` which, as can be seen in Figure 2(d), is clearly favored when tweets are stored in order.

Regarding read-only operations, the adopted data placement strategy may have an high impact on latency, see Figures 2(d) through 2(g).

The `statuses_user_timeline` operation (Figures 2(d)) is mainly composed by a range query (which retrieves a set of the more recent tweets of the user) and is therefore best served when `tweets` are (chronologically) ordered minimizing this way the number of nodes contacted. Taking advantage of SFC's locality preserving property grouping by tags still considerably outperforms the random strategy before saturation.

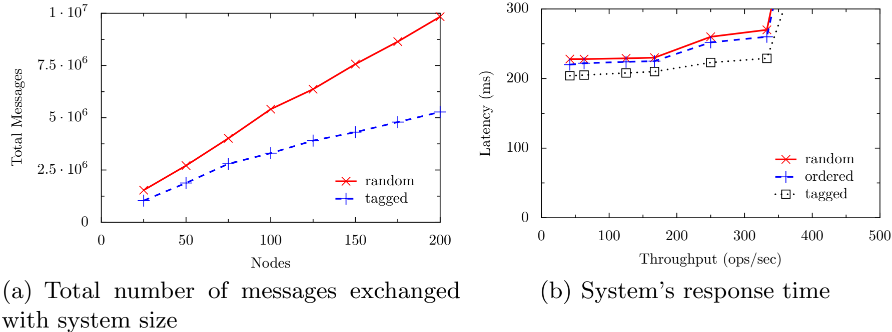
Operations `status_mentions` and `search_contains_hashtag` are essentially correlated searches over `tweets`, by user and by topic, respectively. Therefore, as expected, they perform particularly well when the placement of `tweets` uses the tagged strategy. For `status_mentions` the tagged strategy is twice as fast as the others, and for `search_contains_hashtag` keeps a steady response time up to ten thousand ops/sec while with the other strategies the systems struggle right from the beginning.

Operation `statuses_friends_timeline` accesses the `tweets` collection directly by key and sparsely. To construct the user's timeline the operation gets the user's tweets list entry from `timelines` and for each tweetid reads it from `tweets`. These end up being direct and ungrouped (i.e.. single item) requests and, as depicted in Figure 2(g) best served by the random and ordered placements.

Figure 2(h) depicts the response time for the combined workload. Overall, the new SFC based data placement strategy consistently outperforms the others with responses 40% faster.

Finally, it is worth noticing the substantial reduction of the number of exchanged messages attained by using the tagged strategy. Figure 3(a) compares the total number of messages exchanged when using the random and tagged

strategies. This reduction is due to the restricted number of contacted nodes by the tagged strategy in multi-item operations.



(a) Total number of messages exchanged with system size

(b) System's response time

Fig. 3. Additional evaluation results

Real setting Figure 3(b) depicts the response time for the combined workload in the real setting. The results in the real setting confirm the previous results from the simulated setting. Overall, the new SFC based data placement strategy consistently outperforms the others.

The additional response time in the real setting, compared with the simulated setting, is due to the use of a persistent storage.

5 Related Work

There are several emerging decentralized key-value stores developed by major companies like Google, Yahoo, Facebook and Amazon to tackle internal data management problems and support their current and future Cloud services. Google's BigTable[4], Yahoo's PNUTS[6], Amazon's Dynamo[8] and Facebook's Cassandra[17] provide a similar service: a simple key-value store interface that allows applications to insert, retrieve, and remove individual items. BigTable, Cassandra and PNUTS additionally support range access in which clients can iterate over a subset of data. DataDroplets extends these systems' data models with tags allowing applications to run more general operations by tagging and querying correlated items.

These systems define one or two data placement strategies. While Cassandra and Dynamo use a DHT for data placement and lookup, PNUTS and BigTable have special nodes to define data placement and lookup. Dynamo just implements a random placement strategy based on consistent hashing. Cassandra supports both random and ordered data placement strategies per application but only allows range queries when using ordered data placement. In PNUTS, special nodes called routers, maintain an interval mapping that divides the overall space into intervals and define the nodes responsible for each interval. It also

supports random and ordered strategies, and the interval mapping is done by partitioning the hash space and the primary key's domain, respectively. BigTable only supports an ordered data placement. The items' key range is dynamically partitioned into *tablets* that are the unit for distribution and load balancing. With only random and ordered data placement strategies, existing decentralized data stores can only efficiently support single item operations or range operations. However, some applications, like social networks, need frequently to retrieve general multi correlated items.

Our novel data placement strategy that allows to dynamically correlate items is based on Space Filing Curves(SFCs). SFCs had been used to process multi-dimensional queries in P2P systems. PHT [5] applies SFC indexing over generic multi hop DHTs, while SCRAP [10] and Squid [20] apply SFC indexing to Skip graphs [1] and Chord [22] overlay, respectively.

While in all other systems the multi-dimensional queries are based on predefined keywords and keywords set DataDroplets is more flexible, allowing free form tags and tags set. Therefore, tags are used by applications to dynamically define items correlation. Additionally, in DataDroplets the SFC based strategy is combined with a random placement strategy and with a generic load balancing mechanism to improve uniformity even when the distribution of tags is highly skewed.

Moreover, while all other systems using SFCs in P2P systems run over multi hop DHTs, our tagged data placement strategy runs over a one hop DHT. Therefore, while in these other systems the query processing is done recursively over several nodes in the routing path, with increasing latency, our strategy allows the node that receives the query to locally know the nodes that need to be contacted to answer the query.

6 Conclusion

Cloud Computing and unprecedented large scale applications, most strikingly, social networks such as Twitter, challenge tried and tested data management solutions and call for a novel approach. In this paper, we introduce DataDroplets, a key-value store whose main contribution is a novel data placement strategy based on multidimensional locality preserving mappings. This fits access patterns found in many current applications, which arbitrarily relate and search data by means of free-form tags, and provides a substantial improvement in overall query performance. Additionally, we show the usefulness of having multiple simultaneous placement strategies in a multi-tenant system, by supporting also ordered placement, for range queries, and the usual random placement.

Finally, our results are grounded on the proposal of a simple but realistic benchmark for elastic key-value stores based on Twitter and currently known statistical data about its usage. We advocate that consensus on benchmarking standards for emerging key-value stores is a strong requirement for repeatable and comparable experiments and thus for the maturity of this area. This proposal is therefore a first step in that direction.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 384–393. SODA '03, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003), http://portal.acm.org/citation.cfm?id=644108.644170
2. Boyd, D., Golder, S., Lotan, G.: Tweet tweet retweet: Conversational aspects of retweeting on twitter. In: Society, I.C. (ed.) Proceedings of HICSS-43 (January 2010)
3. Butz, A.R.: Alternative algorithm for hilbert's space-filling curve. IEEE Trans. Comput. 20(4), 424–426 (1971)
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 205–218. USENIX Association, Berkeley, CA, USA (2006)
5. Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., Hellerstein, J.: A case study in building layered DHT applications. In: SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 97–108. ACM, New York, NY, USA (2005)
6. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (2008)
7. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA (December 2004)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. ACM, New York, NY, USA (2007)
9. Galuba, W., Aberer, K., Despotovic, Z., Kellerer, W.: Protopeer: From simulation to live deployment in one step. In: Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on. pp. 191–192 (Sept 2008)
10. Ganesan, P., Yang, B., Garcia-Molina, H.: One torus to rule them all: multi-dimensional queries in p2p systems. In: WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases. pp. 19–24. ACM, New York, NY, USA (2004)
11. Garg, A.K., Gotlieb, C.C.: Order-preserving key transformations. ACM Trans. Database Syst. 11(2), 213–234 (1986)
12. Gupta, A., Liskov, B., Rodrigues, R.: Efficient routing for peer-to-peer overlays. In: First Symposium on Networked Systems Design and Implementation (NSDI). San Francisco, CA (Mar 2004)
13. Java, A., Song, X., Finin, T., Tseng, B.: Why we twitter: understanding microblogging usage and communities. In: WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis. pp. 56–65. ACM, New York, NY, USA (2007)
14. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot

spots on the world wide web. In: STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 654–663. ACM, New York, NY, USA (1997)
15. Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. pp. 36–43. ACM, New York, NY, USA (2004)
16. Krishnamurthy, B., Gill, P., Arlitt, M.: A few chirps about twitter. In: WOSP '08: Proceedings of the first workshop on Online social networks. pp. 19–24. ACM, New York, NY, USA (2008)
17. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. In: SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS) 2009. Big Sky, MT (Ocotber 2009)
18. Risson, J., Harwood, A., Moors, T.: Stable high-capacity one-hop distributed hash tables. In: ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications. pp. 687–694. IEEE Computer Society, Washington, DC, USA (2006)
19. Sagan, H.: Space-Filling Curves. Springer-Verlag, New York (1994)
20. Schmidt, C., Parashar, M.: Flexible information discovery in decentralized distributed systems. In: HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing. p. 226. IEEE Computer Society, Washington, DC, USA (2003)
21. Sousa, A., Pereira, J., Soares, L., Jr., A.C., Rocha, L., Oliveira, R., Moura, F.: Testing the Dependability and Performance of Group Communication Based Database Replication Protocols. In: International Conference on Dependable Systems and Networks (DSN'05) (june 2005)
22. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. pp. 149–160 (2001)
23. Vilaça, R., Cruz, F., Oliveira, R.: On the expressiveness and trade-offs of large scale tuple stores. In: Meersman, R., Dillon, T., Herrero, P. (eds.) On the Move to Meaningful Internet Systems, OTM 2010, Lecture Notes in Computer Science, vol. 6427, pp. 727–744. Springer Berlin / Heidelberg (2010)
24. Vilaça, R., Oliveira, R., Pereira, J.: A correlation-aware data placement strategy for key-value stores. Tech. Rep. DI-CCTC-10-08, CCTC Research Centre, Universidade do Minho (2010), http://gsd.di.uminho.pt/members/rmvilaca/papers/ddtr.pdf
25. Xiongpai, Q., Wei, C., Shan, W.: Simulation of main memory database parallel recovery. In: SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference. pp. 1–8. Society for Computer Simulation International, San Diego, CA, USA (2009)
26. Yu, H., Gibbons, P.B., Nath, S.: Availability of multi-object operations. In: NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation. pp. 16–16. USENIX Association, Berkeley, CA, USA (2006)
27. Zhong, M., Shen, K., Seiferas, J.: Correlation-aware object placement for multi-object operations. In: ICDCS '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems. pp. 512–521. IEEE Computer Society, Washington, DC, USA (2008)