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

Ricardo Vilaça, Rui Oliveira and José Pereira

{rmvilaca,rco,jop}@di.uminho.pt

Techn. Report 08

2010

Computer Science and Technology Center
Departamento de Informática da Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://cctc.di.uminho.pt/

08

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

by Ricardo Vilaça, Rui Oliveira and José Pereira

Abstract

Key-value stores hold the bulk of the data produced by the unprecedented activity of social networking applications. Their scalability and availability requirements often outweigh sacrificing richer data and processing models, and even elementary data consistency.

In this report 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.

To this end, we present the prototype of an elastic key-value data store embodying a novel data placement strategy based on multidimensional locality-preserving mappings. The system is built on the basis of an efficient peer-to-peer overlay, provides atomic access to tuples and flexible data replication.

We evaluate different data placement strategies under the workload of a typical social network application and show that the proposed correlation-aware data placement offers a major improvement on the system's overall response time and network requirements. The elasticity of the system is also put under test by measuring the impact of a significant change on the number of nodes of the system.

1 Introduction

Until now, relational database management systems have been the key technology to store and process structured data. However, these systems based on highly centralized and rigid architectures are facing a major challenge: The volume of data currently quadruples every eighteen months while the available performance per processor only doubles in the same time period [21]. This is the breeding ground for a new generation of elastic data management solutions, that can scale both in the sheer volume of data that can be held but also in how required resources can be provisioned dynamically and incrementally [7, 5, 4, 16]. Furthermore, the underlying business model supporting these efforts requires the ability to simultaneously serve and adapt to multiple tenants with diverse performance and dependability requirements which add to the complexity of the whole system. These first generation remote storage services are built by major Internet players, like Google, Amazon, FaceBook and Yahoo, by embracing the cloud computing model.

DHTs running over tens to hundreds of nodes in a controlled environment with a reasonably stable membership are well suited for the access and management layer of these elastic and dependable data storages [25]. In these data intensive applications, the routing overlay implicit in the DHT is used to find the nodes responsible for a particular data item. At this scale, it is preferable to maintain information about all nodes using one-hop protocols to reduce lookup latency [6], as typical multi-hop DHTs impose a higher cost in routing and searching of data. On these controlled environments of stable and high capacity networks, one-hop DHTs also consume less query bandwidth [17], which is crucial in data intensive systems.

However, to the best of our knowledge, data placement strategies in existing key-value stores [4, 5, 7, 16] efficiently support only single item or range queries. Most applications have however general multi-item operations that request reads and/or writes to a specific subset of items to accomplish a certain customer task while the availability and performance of multi-items operations are highly affected by the data placement strategy [27]. Also, correlation, the probability of a pair of items being requested together in a query is not uniform but often highly skewed [29]. Additionally, correlation is mostly stable over time for real applications. Therefore, if the data placement strategy places correlated items on the same node the communication overhead for multi-items operations tends to reduce.

The challenge here is achieving 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 placement strategy that considers 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 elastic data storage system with a one-hop DHT supporting a conflict-free strongly consistent data with in-place processing capabilities. DataDroplets is suitable to handle multi-tenant data and is meant to be run in a cloud computing environment over tens to hundreds of nodes in a controlled environment with a reasonably stable membership. It extends first generation remote storage services data models with tags allowing applications to mark and query correlated items.

DataDroplets supports also traditional random and order based strategies used in first generation remote storage services. This allows it to adapt and to be optimized to the clients' 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.

Finally, we have evaluated our proposals with a realistic environment and workload based on Twitter in different configurations regarding replication, and number of nodes. The results show that the novel strategy has lower overall latency than others strategies and attest the performance and scalability of DataDroplets.

The remainder of the report is organized as follows. Section 2 presents DataDroplets. Section 3 presents a thorough evaluation of the three data placement strategies. Section 4 discusses related work and Section 5 concludes the report.

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 tuples and the flexible establishment of arbitrary relations among tuples.

A salient aspect of DataDroplets is the multi-item access that allows to efficiently store and retrieve 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 a major drive in the design of DataDroplets and the main contribution of the current work.

2.1 Data placement strategies

DataDroplets builds on the Chord [23] structured overlay network. Physical nodes are kept organized on a logical ring overlay. 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 DHT is expected to provide a uniform distribution of data across all nodes (for the sake of load balancing) while being able to embody specific grouping strategies justified by the current application and workload.

In the following we describe the current data placement strategies offered by DataDroplets. The first is the *random placement*, the basic load-balancing strategy present in most DHTs [23, 18, 28, 11]. The *ordered placement* takes into account order relationships among items favouring the response to *range* oriented reads. The *tagged placement* is our solution to efficiently handle dynamic multi-dimensional relationships of aribitrarily tagged items.

2.1.1 Random placement

The random strategy is based on a consistent hash function [13]. When using consistent hashing each item has a numerical ID (between 0 and MAXID) obtained by, for example, by pseudo-randomly 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 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.

2.1.2 Ordered placement

The ordered strategy places items according to the partial order of the items' keys. 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 [10] 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 $hashorder(o1) < hashorder(o2)$.

In order to make the hash function uniform as well it is needed some knowledge on the distribution of the item's keys [10]. For a uniform and efficient distribution it is needed 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 on, we use a more general approach to achieve load balance.

2.1.3 Tagged placement

The tagged strategy realizes the data placement according to the set of tags defined per item allowing to efficiently support complex tag searches. 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 their tags are lexicographically close. Tags can be viewed as base-n numbers, for example n can be 10 for numeric keywords or 26 if the tags are words. Not all combinations of characters represent meaningful tags, resulting in a sparse tag space with non uniformly distributed clusters of data items.

This mapping is derived from a locality-preserving mapping called Space Filling Curves (SFCs) [19]. An SFC is a continuous mapping from a d-dimensional space to a one-dimensional space ($f : N^d \to N$). 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. Any range query or query composed of tags can be mapped to a set of regions in the tag space and corresponding clusters in the SFC.

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. The position inside the partition is defined by a random strategy.

2.2 Overlay management

In DataDroplets each node maintains complete information about the overlay membership as in [11, 17]. This fits our informal assumptions about the size and dynamics of target environments, tens to hundreds of nodes with a reasonably stable membership, and allows efficient one-hoping routing of requests [11].

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 [11].[1]

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 pro-

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

posed in [14]. Roughly, the algorithm has a node, chosen at random, periodically contacting 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. Read operations leverage replicated data to improve performance. As in Chord [23], we replicate the whole data held by a node into its r successors in the ring, where r is the previously configured replication degree. Replication is done using a synchronous primary-backup algorithm: the completion of a write operation requires the previous update of all replicas. The synchrony requirement can, of course, be released should the application semantics permit.

In the case of a node failure fail-over to one its backups is done as transparently and automatically as possible. While current requests being handled by the failed node are lost any incoming read requests are immediately handled by one of the replicas. Write requests wait for the membership change.

2.3 Request handling

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. The data placement strategy is defined on a collection basis.

The system supports common single item operations such as put, get and delete, multi item put and get operations, and set operations to retrieve ranges and equally tagged items. The detailed set of operations is the following:

```
put(Comparable key, Object value, Set<String> tags)
Object get(Comparable key)
Object delete(Comparable key)
multiPut(Map<Comparable key, Pair<Object, Set<String>>> map)
Map<Comparable, Object> multiGet(Set<Comparable> keys)
Map<Comparable, Object> getByRange(Comparable min, Comparable max)
Map<Comparable, Object> getByTags(Set<String> tags)
```

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 explicitely identify the item by key, `get`, `multiGet` and `delete` the node responsible for the data can be directly identified when the collection is being 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 being distributed with the ordered and tagged strategies, respectively. Otherwise, all nodes need to be contacted and need to process the request.

The processing of requests also needs to take into account data replication. When using replication the replicas may be exploited to improve the read performance. Indeed, in DataDroplets a single read of an item is sent at random to one of the nodes holding the data.

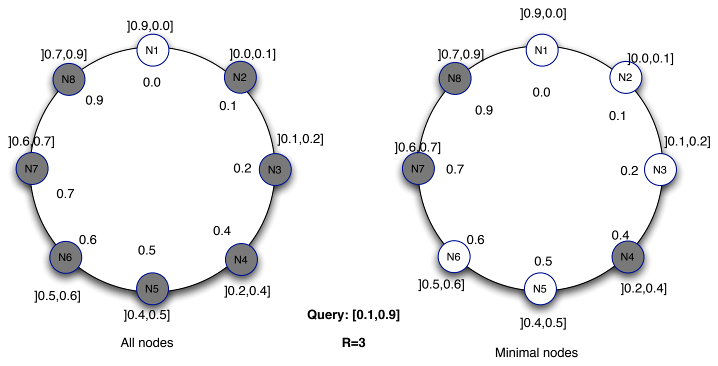


Figure 1: Minimizing the number of contacted nodes

Additionally, for multi item operations, as multiple nodes have redundant data, the number of contacted nodes can be minimized.

Consider the example in Figure 1. In this scenario there are 8 nodes in the system and 3 replicas per item. A getByRange request for all keys in the interval $[0.1, 0.9]$ is issued. If processing is restricted to the primary node of the items then the set of nodes contacted is $[N2, N3, N4, N5, N5, N7, N8]$.

However, making use of the several replicas the set of contacted nodes can be reduced to $[N4, N7, N8]$. Nodes $N4$ and $N7$ will answer with the all stored items, those for which they are the primary node and those for which they are replicas at distance 1 and 2. Node $N8$ will only answer with the items for which it is the primary node.

3 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, then for the impact of adding replication in order to increase fault tolerance and finally, for the effects of scale by substantially increasing the number of nodes.

3.1 Test workload

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

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

The central feature of Twitter is the user *timeline*. A user's timeline is the stream of tweets from the users she *follows* and from her own. Tweets are free form strings up to 140 characters. Tweets may contain two kinds of tags, user mentions formed by a user's id preceded by @ (e.g.. @john) and hashtags, arbitrary words preceded by # (e.g.. #topic) meant to be the target of searches for related tweets.

Our workload definition has been shaped by the results of recent studies on Twitter [12, 15, 2]. In particular, we consider just the subset of the seven most used operations from the Twitter API [24] (Search and REST API as of March 2010):

`List<Tweet>statuses_user_timeline(String userID, int s, int c)` retrieves from userID's tweets, in reverse chronological order, up to c tweets starting from s (read only operation).

`List<Tweet>statuses_friends_timeline(String userID, int s, int c)` retrieves from userID's timeline, in reverse chronological order, up to c tweets starting from s. This operation allows to obtain the a user's timeline incrementally (read only operation).

`List<Tweet>statuses_mentions(String userID)` retrieves the most recent tweets mentioning userID's in reverse chronological order (read only operation).

`List<Tweet>search_contains_hashtag(String topic)` searches the system for tweets containing topic as hashtag (read only operation).

`statuses_update(Tweet tweet)` appends a new tweet to the system (update operation).

`friendships_create(String userID, String toStartUserID)` allows userID to follow toStartUserID (update operation).

`friendships_destroy(String userID, String toStopUserID)` allows userID to unfollow toStopUserID (update operation).

For the implementation of the test workload we consider a simple data model of three collections: `users`, `tweets` and `timelines`. The `users` collection is keyed by userid and for each user it stores profile data (name, password, and date of creation), the list of the user's followers, a list of users the user follows, and the user's tweetid, an increasing sequence number. The `tweets` collection is keyed by a compound of userid and tweetid. It stores the tweets' text and date, and associated user and topic tags if present. The `timelines` collection stores the timeline for each user. It is keyed by userid and each entry contains a list of pairs (tweetid, date) in reverse chronological order.

In a nutshell, the operations listed above manipulate these data structures as follows. The `statuses_update` operation reads and updates the user's current tweet sequence number from `users`, appends the new tweet to `tweets` and updates the timeline for the user and each of the user's follower in `timelines`. The `friendships_create` and `friendships_destroy` operations update the involved users records in `users` and recomputes the follower's `timelines` adding or removing the most recent tweets from the followed, or unfollowed, user. Regarding the read only operations, `statuses_friends_timeline` simply accesses the specified user timeline record in `timelines`, `statuses_user_timeline` accesses a range of the user's tweets, and `statuses_mentions` and `search_contains_hashtag` the `tweets` collection in general.

For the experiments described in the next section the application is firstly initialized with a set of users (that remains unchanged throughout the experiments), a graph of follow relationships and a set of tweets.

Twitter's network belongs to a class of scale-free networks and exhibit a small world phenomenon [12]. As such, the set of users and their follow relationships are determined by a directed graph created with the help of a scale-free graph generator [1].

In order to fulfill `statuses_user_timeline`, `statuses_friends_timeline` and `statuses_mentions` requests right from the start of the experiments, the application is populated with initial tweets. The generation of tweets, both for the initialization phase and for the workload, follows a couple of observations over Twitter traces [15, 2]. First, the number of tweets per user is proportional to the user's followers [15]. 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 [12].

Each run of the workload consists of a specified number of operations. The next operation is randomly chosen taking into account the probabilities of occurrence depicted in Table 1. To our knowledge, no statistics about the particular occurrences of each of the Twitter operations are publicly available. The figures of Table 1 are biased towards a read intensive workload and based on discussions that took place during Twitter's Chirp conference (the Twitter official developers conference, e.g.. `http://pt.justin.tv/twitterchirp/b/262219316`).

3.2 Experimental Setting

We evaluate our implementation of DataDroplets using the ProtoPeer toolkit [8] to simulate 100 and 200 nodes networks. ProtoPeer is a toolkit for rapid distributed systems prototyping that allows switching between event-driven simulation and live network deployment without changing any of the application code.

From ProtoPeer we have used the network simulation model and extended it with simulation models for CPU as per [26]. The network model was configured

Table 1: Probability of Operations

Operation	Probability
search_contains_hashtag	15%
statuses_mentions	25%
statuses_user_timeline	5%
statuses_friends_timeline	45%
statuses_update	5%
friendships_create	2.5%
friendships_destroy	2.5%

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 [22]. 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. Each node was configured and calibrated to simulate one dual-core AMD Opteron processor running at 2.53GHz.

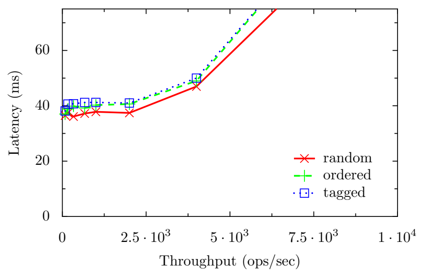
For all experiments presented next the performance metric has been the average request latency as perceived by the clients. A total of 10000 concurrent users were simulated (uniformly distributed by the number of configured nodes) and 500000 operations were executed per run. Different request loads have been achieved by varying the clients think-time between operations. Throughout the experiments no failures were injected.

3.3 Evaluation of data placement strategies

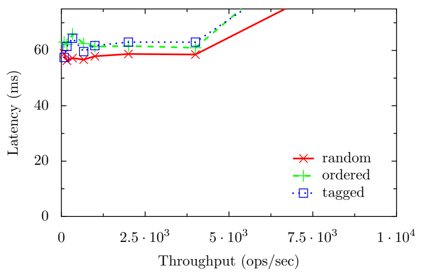
The graphs in Figure 2 depict the performance of the system when using the different placement strategies available. The workload has been firstly configured to only use the random strategy (the most common in existing key-value stores), then configured to use the ordered placement for both the `tweets` 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.

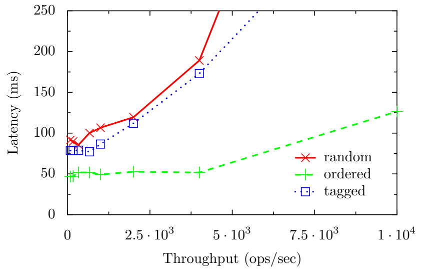
We can start by seeing that for write operations (`statuses_update` and `friendships_destroy`) the system's response time is very similar for all scenarios (Figures 2(a)and 2(b)). Both operations read *one* user record and subsequently add or update one of the tables. The costs of these operations is



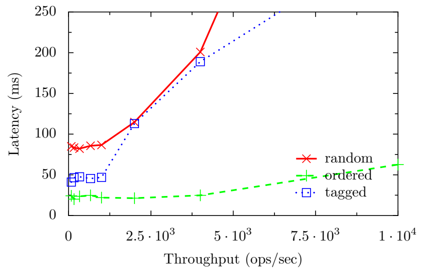
(a) statuses_update op



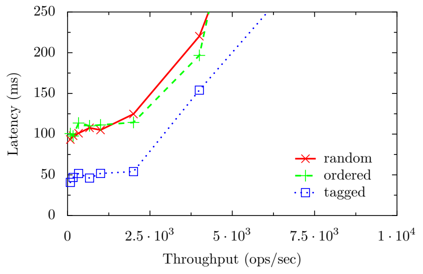
(b) friendships_destroy op



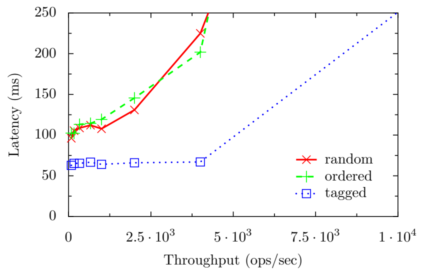
(c) friendships_create op



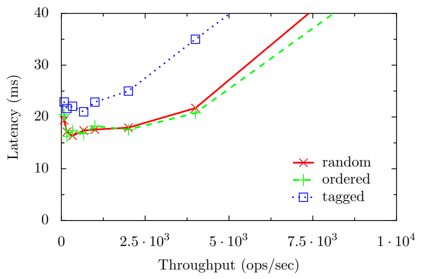
(d) statuses_user_timeline op



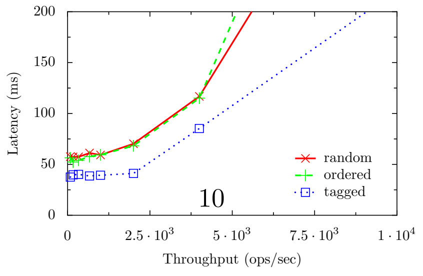
(e) statuses_mentions op



(f) search_contains_hashtag op



(g) statuses_friends_timeline op



(h) Overall workload

Figure 2: System's response time for Twitter workload with 100 nodes

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 compares the total number of messages exchanged when using the random and tagged strategies.

3.4 Evaluation of node replication

Data replication in DataDroplets is meant to provide fault tolerance to node crashes and improve read performance through load balancing. Figure 4 shows the results of the combined workload using tagged placement when data is replicated over three nodes.

The *minimized* and *not minimized* lines correspond to a synchronous replication algorithm where the write operations completion depends on the successful writes in all replicas. As explained in Section 2.3, the minimized strategy takes advantage of replication to minimize the number of nodes contacted per operation having each node responding for all the data it holds, while the other, on

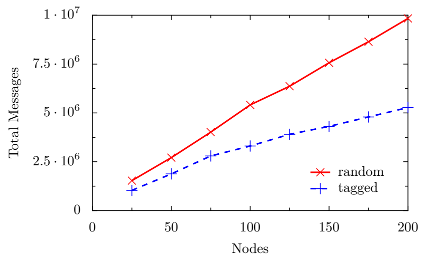


Figure 3: Total number of messages exchanged with system size

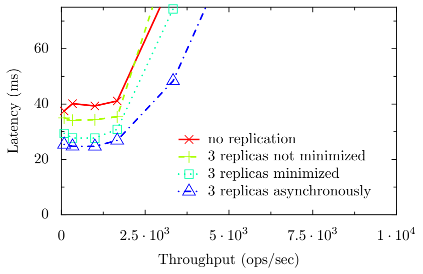


Figure 4: Impact of various replication techniques

the contrary, leverages replication to increase concurrent accesses to different nodes.

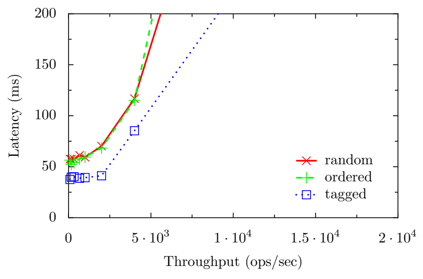
In both cases we can see that despite the impact synchronous replication inevitably has on write operations, the overall response time is improved by 12% with *not minimized* strategy and 27% with *minimized* strategy.

To better assess the benefits of replication to the performance of read operations we also tested the system with asynchronous replication. The overall gain is up to 14% which would not, per se, justify the increased complexity of the system. It is actually the dependability facet that matters most, allowing to provide seamless fail over of crashed nodes.

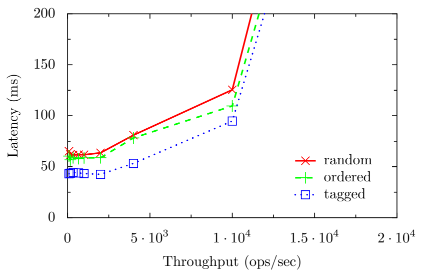
3.5 Evaluation of the system elasticity

To assess the system's response to a significant scale change we carried the previous experiments over the double of the nodes, 200. Figure 5(b) depicts the results.

Here, it should be observed that while the system appears to scale up very well providing almost the double of throughput before getting into saturation, for a small workload, up to 2000 ops/sec with 200 nodes there is a slightly higher latency. This result motivates for a judicious elastic management of the system to maximize performance, let alone economical and environmental reasons.



(a) 100 nodes configuration



(b) 200 nodes configuration

Figure 5: System's response time with 100 and 200 nodes.

4 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[5], Amazon's Dynamo[7] and Facebook's Cassandra[16] 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 marking 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 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 store and retrieve correlated items is based on Space Filing Curves(SFCs). SFCs have been used for the placement of static and pre-defined multi attribute items and multi attribute queries in [20, 9]. These systems use a similar approach to index data and distribute the index at nodes. However, their query engines differ. The approach used by Ganesan et. al. [9] increases false-positives (e.g., non-relevant nodes may receive the query) to reduce the number of sub-queries while Squid [20] decentralize and distribute the query processing across multiple nodes in the system.

Differently from other systems that use SFC strategies [20, 9] for multi attribute queries DataDroplets allows the application to dinamicaly define tags as a hint to the items correlation, using them to construct the SFC multi dimensional space and then efficiently querying correlated items. Furthermore, other systems distribute the query processing while we only use the SFC to reduce the overall one dimensional index space to a set of partitions and redirect the query to the nodes responsible for that partitions. Additionally, in DataDroplets the SFC based strategy is combined with an generic load balancing mechanism to improve uniformity even when the distribution of tags is highly skewed.

5 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 report, 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. As a secondary contribution, 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 this direction.

References

[1] Barabási, A.L., Bonabeau, E.: Scale-free networks. Scientific American 288, 60–69 (2003)

[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), `http://portal.acm.org/citation.cfm?id=1298455.1298475`

[5] 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)

[6] Cruces, N., Rodrigues, R., Ferreira, P.: Pastel: Bridging the gap between structured and large-state overlays. In: CCGRID. pp. 49–57 (2008)

[7] 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)

[8] 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)

[9] 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)

[10] Garg, A.K., Gotlieb, C.C.: Order-preserving key transformations. ACM Trans. Database Syst. 11(2), 213–234 (1986)

[11] 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)

[12] 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)

[13] 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)

[14] 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)

[15] 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)

[16] 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), `http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf`

[17] 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)

[18] Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg. pp. 329–350. Springer-Verlag, London, UK (2001)

[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] Skillicorn, D.: The case for datacentric grids. Tech. Rep. ISSN-0836-0227-2001-451, Department of Computing and Information Science, Queen's University (November 2001)

[22] 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)

[23] 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)

[24] Twitter: Twitter api documentation. http://apiwiki.twitter.com/Twitter-API-Documentation (2010)

[25] Vilaça, R., Oliveira, R.: Clouder: a flexible large scale decentralized object store: architecture overview. In: WDDDM '09: Proceedings of the Third Workshop on Dependable Distributed Data Management. pp. 25–28. ACM, New York, NY, USA (2009)

[26] 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)

[27] 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)

[28] Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley (Apr 2001), `citeseer.ist.psu.edu/zhao01tapestry.html`

[29] 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)