Key-Key-Value Stores for Efficiently Processing Graph Data in the Cloud

Alexander G. Connor, Panos K. Chrysanthis, Alexandros Labrinidis
Advanced Data Management Technologies Laboratory
Department of Computer Science, University of Pittsburgh
{agc7,panos,labrinid}@cs.pitt.edu

***Abstract*— Modern cloud data storage services have powerful capabilities for data-sets that can be indexed by a single key – key-value stores – and for data-sets that are characterized by multiple attributes (such as Google's BigTable). These data stores have non-ideal overheads, however, when graph data needs to be maintained; overheads are incurred because related (by graph edges) keys are managed in physically different host machines. We propose a new distributed data-storage paradigm, the key-key-value store, which will extend the key-value model and significantly reduce these overheads by storing related keys in the same place. We provide a high-level description of our proposed system for storing large-scale, highly interconnected graph data – such as social networks – as well as an analysis of our key-key-value system in relation to existing work. In this paper, we show how our novel data organization paradigm will facilitate improved levels of QoS in large graph data stores.**

I. Introduction

Cloud computing is a broad class of techniques and paradigms that reflect large-scale, distributed computing and storage as a service. Customers of these cloud services pay per use of storage space, bandwidth and processor time; the rate is often determined by a *Service-Level Agreement* that specifies the fee as a function of response time or availability for requests [1]. Such services are used internally within computing firms as well; in this setting, consumer-facing applications use the internal cloud services for fulfilling business needs [1], [2].

Some of the more well-known internal cloud systems include Dynamo of Amazon.com [1], PNUTS of Yahoo! [2] and BigTable from Google [3], which use key-value stores. Key-value stores are data structures wherein there is a primary key and an object; the object can only be referred to by the key. The creators of Dynamo and PNUTS were motivated by the observation that most of the data that applications work with are indexed only by a primary key and most often are accessed or modified one item at a time [1], [2]. BigTable manages more complex data structures by allowing values to have many attributes – these are still identified by only one key. G-Store [4] and ecStore [5] are University prototypes of transactional key-value stores that support multiple key manipulations with strict consistency.

Motivating Example: Emerging cloud services often work with requests for data identified by two keys – examples include Facebook (facebook.com) and Twitter (twitter.com). Consider Twitter, a social networking service where users send status updates and read the status updates of their friends, or people they *follow*. Twitter thus needs to be able to specify this user to user relation: `(user1, user2, relationship)`. For our example, suppose that *relationship* is `follows` and `alice` is the user who follows `bob`. The relationship could also be something else, such as `blocked`, where the first user is not allowed to see the second user's updates.

Users need to be able to ask queries such as *who follows* `bob`*?* and *who does* `alice` *follow?* To implement these relations – the *follows-list* and the *followers-list* – in a key-value store, both lists need to be maintained for both `alice` and for `bob`. If only the follows-list was maintained, the follows-list for every user in the graph would need to be examined when determining the followers-list for `bob` – this means a high-cost join. One could argue that edges of the graph could be indexed in an intelligent fashion; even with the best indexing the lists for `alice` and `bob` could still be stored on separate machines. This creates an overhead in performance or a potential inconsistency in the data [6].

We propose to unify these four lists by storing data as a `(source-key, target-key, value)` relation; now, if both `alice` and `bob`'s data are kept in the same place, then both queries can be answered from *one* site. Our proposed approach reduces the overhead of storage by eliminating duplicate records and reduces performance overhead and improves consistency by keeping related records in one site.

Unlike Pregel, a paradigm for distributed graph data processing [7], our proposed system provides robust graph data storage. Similar to cloud key-value stores, our system will provide durable, available and fast key-value and key-key-value data management with flexible consistency guarantees. Our system could be used for implementing paradigms like Pregel, and a variety of graph operations such as reachability queries, shortest path queries and subgraph matching queries. Applications performing such queries will be able to use our key-key-value system's API for effecient processing.

Contributions: In this work we contribute the following to the literature:

- A novel model for scalable graph data stores that extends the key-value model, the *key-key-value* store (Sec. II).
- A high-level design for the *key-key-value* system (Sec. III).
- A novel on-line algorithm for partitioning graph data in *key-key-value* stores (Algorithm 1, given in Sec. III-F and evaluated in Sec. IV).

978-1-4244-9196-4/11/$26.00 © 2011 IEEE

ICDE Workshops 2011

II. System Model

Our proposed system will have four layers of organization beyond the general cloud infrastructure:

1) *Physical Layer* – We will assume that machines have heterogeneous capabilities and are interconnected in a network where each machine can reach every other machine most of the time.
2) *Logical Layer* – Here, each node is not actually a physical machine; it is a virtual machine. The services hosted at each virtual node will see the rest of the system as a grid; a global distributed transactional table maps virtual nodes to physical machines. In this way we can (1) maintain a perfect grid, no matter how many machines we have, (2) give more powerful machines heavier loads than weaker machines by assigning more virtual nodes and (3) add or remove machines from the system without disrupting the logical organization. The grid structure allows our system to organize key-key-values with the on-line partitioning algorithm, as well as support efficient global communication.
3) *Address Table* – Our system maps individual keys to rows and columns in the grid; this mapping cannot be made by a hash function – it has to be stored. Our system thus uses a distributed hash table – the Address Table – similar to the hash table used by Dynamo [1].
4) *Application Layer* – User applications, also referred to as *users*, see the entire system as a black box where basic I/O operations are supported. Our system provides an API for key-key-values, in addition to key-values.

We will discuss in the next section each layer and the specifics of system structure and function at each layer in detail; the key-key-value system needs to support a specific set of operations at each layer beyond the key-value operations to uphold the desired characteristics of cloud data stores and interface with the other layers.

III. Implementation Details

A. Application Layer

We begin our discussion with the application layer; this is the layer of the system that outside users interface with. The following user operations are supported:

- `put(source-key,target-key,value)` – Updates the `(source-key,target-key)` specified with `value`. In our example, `put(alice,bob,follows)` is called when `alice` begins following `bob`. Both keys must be non-null. If `value` is null, then the system deletes the record for `(source-key,target-key)`. If the key-pair does not exist in the system, then it is inserted; if one key does not exist in the system and the other does exist, then the new key is inserted as a source-key in the same node as the existing key.
- `get(source-key,target-key)` – Gets the value and versions (as a vector clock [8], [9]) for a given `(source-key,target-key)` pair. If one key is null, then this operation gets the list of records matching the non-null key in the non-null key's position. For example, `get(null,bob)` will return a list of users who follow or who block `bob`, and what relationship each has to `bob`.
- `sync(source-key,target-key)` – Blocks the user application thread until the given key-pair is made consistent with any other node that calls `sync` for the same key-pair. Details on how this works are given in Section III-D.

The key-key-value system also supports key-value operations, including `get(key,value)`, `put(key,value)`, `sync(key,value)`. The mechanics of these are the same as their key-key-value counterparts, the only difference is that these operate on data associated with only one key – preserving the functionality of existing key-value systems.

B. Address Table

As stated above, our goal is to store keys that are related in the same node for performance reasons. In an abstract graph, however, determining which objects to store on which nodes requires some computation. The grid-address, therefore, of each key needs to be stored by our system, so that nodes know where to look for a given key. Each node has a grid-map that associates address-ranges to nodes (and physical machines).

Almost all key-value storage systems use a hash function to determine where to store a specific key-value. This has the benefit of each machine in the distributed system being able to determine the location of a key-value in a local manner. Unfortunately, the hash function will assign key-values to nodes randomly by design, without any regard for how the key-values are connected. In the case of graph data, this means that related keys are likely to be stored in a variety of places. In fact, the larger the set of related keys is, the higher the expected number of different storage sites is. When working with the graph data, an application will thus have to work with several machines – creating overheads with consistency, availability and performance. Our system eliminates such overheads by assigning keys to machines based how they are related, and using the global address table to keep track of their locations.

As the structure of the graph changes and the organization of the grid changes, key addresses will need to change. When this occurs, the Address Table is updated. The Address Table is a distributed transaction-supporting structure; the ACID properties are necessary for preserving the correctness of address-change operations.

C. Cache operations

Each logical node in the network acts as the entry point in the system for user requests. These connections are routed to physical machines arbitrarily by DNS; if a cache for the session already exists on some virtual node, then that node is used for the connection. For each new session, a virtual node is selected by load balancing and a cache is created. The node uses the following operations to pull data into the cache and propagate it out:

- `address(key)` – Gets the grid-address of `key` from the Address Table. Once the location l of the key is established, `(key,` l`)` is inserted into the cache.

- `read(node,source-key,target-key)` – When the user application calls the `get` operation, the cache reads the replica at `node` for `(source-key,target-key)` into the cache. If the cache has available space, the source-key's data and list of target keys is also read.
- `write(node,source-key,target-key,value)` – A background thread in the cache periodically writes modified key-pairs back to the replica at `node`. Note that this occurs periodically whether or not the key-pair has been evicted from the cache, thus it is a *lazy write-through* policy. When the user application calls `sync(source-key,target-key)`, any un-written-through updates to the key-pair are immediately sent to the replicating node. Once this has succeeded, the user thread is unblocked.

The `read` and `write` operations can return with *key not found*. In this case, the cache uses `address` to find the latest location of the requested source-key.

D. Critical Reads and Writes

In the case where user applications are working with data that has to be fresh, our system provides this guarantee through the `sync` operation. In our example, suppose a user connected to node B makes a critical update – `bob` blocking `eve`. Shortly after (in global time), another user thread connected to node E requests a list of all people that have a connection to `eve`. After calling `put(bob,eve,blocked)`, B calls `sync(bob,eve)`. This forces the cache to call `write(`R`,bob,eve,blocked)` for each node R that replicates `(bob,eve)`.

At node E, the user application first calls `sync(bob,eve)`. This forces the application thread to wait until the cache has gotten a fresh copy of the value at `(bob,eve)` from some replica X – as each replicating node has gotten the write from B, E will read the fresh data. Now, the user thread will resume, and get the update that `bob` has blocked `eve`.

Order Consistency: Although using the `sync` operation guarantees consistency, our system has an order consistency guarantee by default: updates made at the same node are always propagated to replicas in the same order. This is a standard functionality of existing cloud key-value stores – our system supports it as well, and supports it for key-key-values. When a user application makes a connection to our system, the system will guarantee that the application is interacting with the same node throughout the session. Once the application has ended the session, this guarantee is voided.

E. Logical Layer

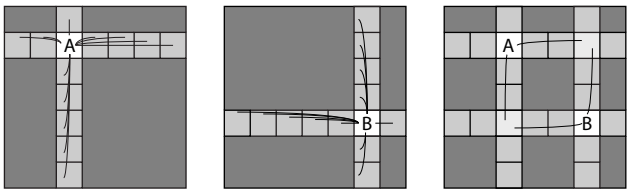
Each virtual node in the logical layer will be responsible for managing a set of *source keys*; each source key has some data associated with it as in a key-value store (in our example, the user's profile and status updates) and a list of *target keys* that the source key links to in the graph. These are stored in the *key-key-value table*; in our example, this would be the table that stores the follows-list for every user whose data is hosted at that node. Each key-pair, furthermore, has a value – in our example, this value can be the *follows* or *blocked* relationship.

Also stored in the same key-key-value table is an inverse mapping of keys; for every source key, each *inverse key* is stored as a `(inverse-key, source-key, value)` tuple. Consider `alice` and `bob`. The tuple `(alice,bob,follows)` is stored in the node that hosts all of `alice`'s data; the *same tuple* is stored in the node that hosts all of `bob`'s data (in his *followers-list*). If both `alice` and `bob` are hosted in the same node, then *only one* `(alice,bob,follows)` *tuple needs to be stored* in that node.

As stated earlier, virtual nodes in our system are mapped to a grid. This grid maintains the logical organization of the network – it is the basis for decisions about where replicas are placed, and where messages are sent to and pulled from. The grid-map is a global distributed structure that stores the `(row-range, column-range, physical-machine)` relation – the *row-range* and *column-range* give the ranges of key addresses; *physical-machine* gives the physical network address of the machine that hosts the virtual node that manages keys with addresses that fall in the given ranges.

The grid-map furthermore supports transactions; this is because some updates to the grid-map involve changing many parts of the map atomically and the grid-map must be consistent for all nodes. Other updates involve changing only one cell of the grid, as we will see later. We posit that during normal systems operations, changes to the grid-map will be infrequent and applications will need information from a limited number of locations in the grid-map in a session – a side-effect of locality by computation (discussed below in Section III-F).

In order to support such transactional operations, most cloud data stores have some unified protocol for communication between nodes. These can cover both sending messages about system state [2] and propogating replicas [1]. Our system will de-couple these; it will propogate replicas passively and use a distributed mutual exclusion protocol (DME) similar to Maekawa's [10] for system state updates.



(a) A sends to all nodes on its line. (b) B reads from all nodes on its line. (c) A's message reaches B through two nodes.

Fig. 1. FPP-based communication in the grid.

Our distributed mutual exclusion scheme (Fig. 1), like Maekawa's, use a finite project plane (or *FPP*). An FPP is a geometric structure characterized by the following axioms: *every point is incident on exactly two lines* and *every line passes through exactly two points*. This is at the core of Maekawa's protocol – a node in the distributed system corresponds to one point and one line in the FPP. This technique requires between $3\sqrt{N}$ and $5\sqrt{N}$ messages to create mutual exclusion – an asymptotic improvement over the previous methods that required $\Theta(N)$ messages [10].

The key-key-value system will use a relaxed version of FPPs in its communication protocols. As stated above, each physical machine in our system can have several virtual nodes, which are organized in a grid. Each virtual node will send system state updates to and request state information from nodes that lie in the same row and column. This ensures the following properties: every point is incident on *at least* two lines and every line passes through *at least* two points. Think of each virtual node representing a point and a line. The *point* that the node corresponds to is its grid cell. The *line* that it corresponds to is the set of virtual nodes that lie on the same row and column. Suppose some node A sends a message to every point-node incident on the A-line. If another node B requests messages from every point-node on its B-line, then the message from A will reach B through *two* point nodes (see Figure 1). In this way, if a failure in one of the connecting nodes occurs, the message will still be passed.

Load Maintenance Operations: Each logical node has a set of operations available to it for completing application requests and for managing load. The load management operations allow a virtual node to split if it is overloaded, and merge with neighbor virtual nodes if all nodes are underloaded. When these operations are performed, they must maintain the structure of the grid (as in Figure 2), and for this reason distributed mutual exclusion is necessary to ensure that a history's splits and merges are serializable for the grid. For instance, if some node X receives two separate requests to split, then it can accept both requests without compromising the grid structure. On the other hand, if X receives a request to split row-wise and merge column-wise, it must choose (based on the logical clock of requests) which request to honor.

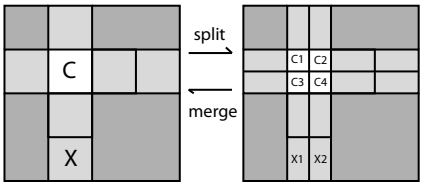


Fig. 2. Splitting and merging a node.

Splitting: Splitting is necessary to prevent overloading; when a node C has load greater than some threshold, it initiates a split. The first step is to contact each node X that lies in the same row or column as C (referred to as the line l, see Figure 2) and determine if a split is possible and to acquire the split-lock. If there is another node Y in l that is involved in a conflicting merge operation, Y contacts the merging node Z to make sure that the merge will succeed. If the merge will succeed, then C must wait to split; otherwise, C tells all nodes in l to split. Once each splitting node has succeeded, then C updates the grid map and releases its split-lock.

Merging: When some virtual node C has load less than a certain threshold (measured by CPU utilization and disk utilization) it contacts each adjacent node C_j to test if a merge is possible (see Figure 2). If each of three C_j responds that it has load less than the threshold, each C_j contacts every node X_k that lies in the same row and column to determine X_k's load and propose a merge. If every X_k that was contacted has load below the threshold, then C tries to acquire the merge-lock. For each set of merging nodes $C_1, ..., C_4$ a node C is created that handles keys in the in unioned key-range of nodes $C_1, ..., C_4$. Each of the nodes $C_1, ..., C_4$ stays on-line, but only as a router that forwards every request to C. All nodes X_1, X_2 in the merging rows and columns similarly merge into new nodes X, as illustrated in Figure 2. Once C has confirmed that the merge has succeeded, it updates the grid-map and the nodes $C_1, ..., C_4$ are notified by C that they can shut down; finally, the merge-lock is released.

In the event that a failure is detected in either process (by timeout), each node that becomes aware of the failure considers the operation cancelled. The current failed operation is reversed and the relevant locks released. Once recovery has occurred and has been confirmed, the originating node C will re-start the operation if necessary.

F. Computation of Locality

Although a key's grid-address is assigned arbitrarily when it is first inserted into a virtual node, the node that it first resides in may not be the optimal place to store that key. Our system, thus, needs to be able to determine reasonably good locations for keys, such that related keys are stored in the same or grid-adjacent virtual nodes. In order to achieve this, we introduce the concept of *computation of locality*.

Spatial data structures, especially grid files, take advantage of spatial locality by storing items that are (spatially) related to one another in the same place, or at worst in a nearby place. Note that traditional grid files achieve this locality by hashing keys to rows and columns in the grid; a cell in the grid represents a set of key-pairs determined by the hash function [11]. For our system, which stores graph data, this scheme will cause related keys to be spread out across whole rows and columns. Instead, our key-key-value system will store all data for some deliberately selected, related keys in one grid cell.

Unfortunately, for abstract structures such as social networks, spatial locality is not intrinsically available for systems to exploit. We propose to use a novel on-line graph partitioning algorithm to determine the grid-addresses of keys. On each node, a continuously-running background process calls the on-line partitioning algorithm (Algorithm 1) on each source-key in an arbitrary order. The structure of the graph changes in the following events, which may lead to the migration of source-keys:

Split Event: When a virtual node splits, each of the source-keys stored in that node must be re-addressed. The node arbitrarily assigns a new grid-address to each key. The keys are then immediately transferred and the Address Table is updated; in this way the split can be completed quickly and the new nodes can begin operations. Once the split has succeeded, the new nodes start up the on-line partitioning algorithm to better organize the recently moved keys.

Merge Event: When a group of nodes merge, the process is much simpler: each source-key is updated with the new node's grid-location intervals. Merging has the benefit of reducing storage overhead in the new node by eliminating duplicate key-key-value entries in the merged nodes. Also, all of the edges that crossed between the merged nodes will be contained in the same node; improving both performance and availability.

Target Key Insert or Delete Event: When a user inserts or deletes a key-pair, a target key is added to or removed from a source key's list. A potentially better arrangement of the source-keys may then exist. If the source key needs to be moved, a copy is made on the new hosting node. The address table is then updated with the new location; finally, the old host node will delete the source-key from its store.

Within each node's data store, we maintain not only the key-key-values and source-key's data, we also maintain some aggregates about each source-key. These include a list of nodes that host the source-key's target-keys and inverse-keys, and the number of target-keys and inverse-keys hosted on each node. Using these counts, our on-line algorithm can quickly determine if a key can be placed in a better location. After each key-pair insertion or deletion, these counts are updated as part of the operation. Our proposed algorithm is similar to that proposed by Zanghi et al [12]; however, their work focused on the on-line addition and removal of graph vertices whereas we are concerned with operations on edges. To our knowledge, the only other similar approaches to ours are those by Sun et al [13], Kernighan and Lin [14], and Fiduccia and Mattheyses [15]; these work in an off-line fashion.

Algorithm 1 On-line algorithm for determining the optimal host virtual node for a source-key.

Input: Source-key s; node H where s is hosted; list $(H', l(H'), c(H'))$ where H' is a node, $l(H')$ is the storage overhead at H', $c(H')$ is a count of target-keys and inverse-keys of s that are hosted on H'; capacity ϵ, threshold δ.
Output: Node H' such that H''s storage requirement if s were hosted on H' is less than some threshold and the number edges that cross between nodes from s is minimal.

Add each H' to a priority queue q, ordered by $c(H')$.
while true **do**
  Let $X \leftarrow \text{top}(q)$; pop q.
  Return H **only if** $c(X) < c(H) + \delta$.
  if $H \neq X, l(X) + \text{storageCost}(s) < \epsilon$ **then**
    Move s to X.
    for each target-key or inverse-key t of s **do**
      Update t's count-list, reflecting s's new location.
      Return X.
    end for
  end if
end while

G. Replication

Each node additionally has a set of key-ranges that it is responsible for replicating. These ranges are defined as the set of keys in the same logical column *or* the same logical row as the node. Suppose that node A is sending copies of data item d to d's replicating nodes. A first sends it to the adjacent nodes in the grid; each of which returns a success or failure message. A counts the number of distinct physical locations that have received a copy, and if that number is less than some threshold W, A sends the update of d to the next adjacent nodes in its row and column. This is repeated until W or more physical copies have been made. Our system, thus, uses $W + R \leq N$ replication, in the interest of greater availability and lower response times [16].

Each logical node has the following replication operations:

- `push(node,source-key,target-key)` – sends the versions for the given key-pair to `node`. This is done periodically in the background; when a node's load is below a certain threshold or when enough time has passed since an update, it begins pushing to its neighbors. After each push, the receiving node will reply indicating success or that it is busy and the push should be performed again later. If the pushing node has not gotten a reply from the receiving node after some period of time, it will assume the receiving node has failed and initiate recovery.
- `pull(node,source-key,target-key)` – requests the versions for the given key-pair from `node`. A node only does this when it first comes on-line or begins replication of the key-pair.

Conflict Resolution: Suppose that some node Z receives two or more conflicting updates d_1 and d_2 of a `(source-key,target-key)`. Each of d_1 and d_2 has a set of versions identified by vector clocks; each clock contains the ID of the virtual node that made the update and that node's local timestamp. Z will merge the versions of d_1 and d_2, eliminating old versions where possible [1], [9].

H. Physical Layer

In any large-scale system, the ability to add and remove hardware resources as demands change and recover gracefully from hardware failures is essential [1]. We subdivide these hardware changes into two categories: *planned* and *unplanned*. Planned changes include incorporating a new machine, taking an old machine off-line or temporarily removing a machine for maintenance and upgrades. Unplanned changes are the result of unexpected hardware or software failures; generally, these lead to a machine becoming isolated from the network – our system utilizes existing well-known distributed and cloud recovery schemes [1], [17], [18] to handle physical failures.

Adding Resources: Adding resources generally means adding a machine to the network. When this is done, the *physical controller* running on the machine starts up and takes over a virtual node from another physical machine. Occasionally, a logical split needs to take place before the migration is complete. Once the migration of a virtual node from one machine to another is complete, the grid-map is updated to reflect the node's new location.

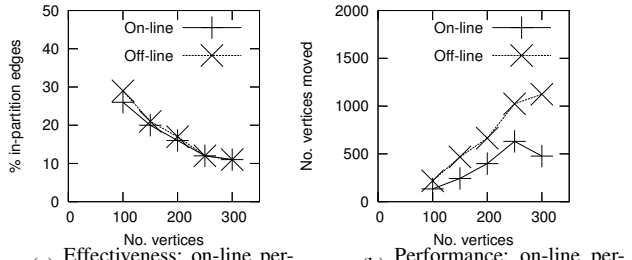
Removing Resources: When a machine needs to be removed from the system, an administrator connects to that machine's physical controller and issues a shutdown command. The machine then tries to migrate all of its virtual nodes to other machines. When this succeeds, it shuts itself down. In some circumstances, this can be done automatically – e.g., the machine senses that it is over-heating or detects disk failures.

IV. Evaluation

In addition to demonstrating the advantages of our proposed system through the theoretical foundations presented earlier, we also briefly present the experimental evaluation of our proposed on-line partitioning algorithm. So as to evaluate our proposed algorithm, we have generated graphs with incoming branching factors between 10% and 100% of the number of vertices in the graph. Branching factors fall on a Zipf distribution with $\alpha = 1.5$. This is to reflect the observation that in social networks, some users have far more followers than others.

In our experiments, we generated three graphs for each evaluated graph size (number of vertices) and fixed the partition size to be at most 30 vertices. We measured the percentage of edges that cross between partitions and the number of vertices moved between partitions for both our proposed algorithm and the off-line algorithm from the work of [15] and [14]. In our implementation of the off-line algorithm, the graph is partitioned hierarchically until all partitions are smaller than the size limit. In each partitioning, the algorithm is allowed to iterate until the increase of the moving average of the improvement from each of the last ten iterations is greater that 99%. For our on-line algorithm, we generate a random sequence of edge insertions and run the algorithm periodically, after every $|V|/10$ insertions where $|V|$ is the number of vertices in the graph.

In Figure 3(a) we see that our proposed on-line algorithm does about as well in terms of partitioning as the off-line algorithm, with the benefit of higher speed and parallelizability. This improvement in performance is apparent in Figure 3(b); where the average number of moved vertices is far less for the on-line algorithm than for the off-line algorithm.



(a) Effectiveness; on-line performs as well as off-line

(b) Performance; on-line performs better than off-line

Fig. 3. Experimental evaluation of our proposed on-line algorithm.

V. Conclusion

We have provided here a high-level description of a system that supports key-key-value stores. Such a system could be deployed as a cloud service that gracefully manages tremendous amounts of graph data, a characteristic of social networks. Through using the principles behind Maekawa's algorithm and through unifying the data structures needed to maintain graph data, our key-key-value system promises to reduce the overheads encountered in a key-value-only store. Experimental evaluation of our on-line partitioning algorithm further supports this promise. Our system is currently in development; we plan to realize our key-key-value system's potential through a complete implementation.

Acknowledgment

We thank D. Cole of our Department's Algorithms group as well as N. Farnan, T.N. Pham and S. Snyder of our group for their helpful feedback.

References

[1] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[2] B. F. 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. VLDB Endow.*, vol. 1, no. 2, 2008, pp. 1277–1288.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proc. 7th USENIX Symp. on Oper. Syst. Design and Impl.*, 2006, pp. 205–218.

[4] S. Das, D. Agrawal, and A. El Abbadi, "G-store: a scalable data store for transactional multi key access in the cloud," in *Proc. 1st ACM symposium on Cloud computing*, 2010, pp. 163–174.

[5] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," in *Proc. VLDB Endow.*, vol. 3, no. 1, 2010, pp. 506–517.

[6] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. 2010 intl. conf. on Management of data*, 2010, pp. 135–146.

[8] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[9] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proc. of the 11th Australian Comp. Sci. Conf.*, 1988, pp. 56–66.

[10] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Computer Syst.*, vol. 3, no. 2, pp. 145–159, 1985.

[11] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 38–71, 1984.

[12] H. Zanghi, C. Ambroise, and V. Miele, "Fast online graph clustering via erdos-rnyi mixture," *Pattern Recognition*, vol. 41, no. 12, pp. 3592 – 3599, 2008.

[13] Y. Sun, J. Han, P. Zhao, Z. Yin, H. Cheng, and T. Wu, "Rankclus: integrating clustering with ranking for heterogeneous information network analysis," in *Proc. EDBT: Advances in Database Technology*, 2009, pp. 565–576.

[14] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, vol. 49, pp. 291–307, 1970.

[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Design Automation Conference*, 1982, pp. 175–181.

[16] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[17] M. T. Ozsu and P. Valduriez, "Distributed database systems: where are we now?" *Computer*, vol. 24, no. 8, pp. 68 –78, Aug. 1991.

[18] M. Choy and A. K. Singh, "Efficient fault tolerant algorithms for resource allocation in distributed systems," in *Proc. ACM symp. on Theory of computing*, 1992, pp. 593–602.