Distributed Range-Based Meta-Data Management for an In-Memory Storage

Florian Klein(✉), Kevin Beineke, and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
Florian.Klein@uni-duesseldorf.de

Abstract. Large-scale interactive applications and online graph processing require fast data access to billions of small data objects. DXRAM addresses this challenge by keeping all data always in RAM of potentially many nodes aggregated in a data center. Such storage clusters need a space-efficient and fast meta-data management. In this paper we propose a range-based meta-data management allowing fast node lookups while being space efficient by combining data object IDs in ranges. A super-peer overlay network is used to manage these ranges together with backup-node information allowing parallel and fast recovery of meta data and data of failed peers. Furthermore, the same concept can also be used for client-side caching. The measurement results show the benefits of the proposed concepts compared to other meta-data management strategies as well as its very good overall performance evaluated using the social network benchmark BG.

1 Introduction

Large-scale interactive applications and online graph processing must often manage billions of small data objects and require low-latency data access. Facebook for example keeps billions of small data objects in around 1,000 memcached servers to support more than one billion users [11]. Around 70 % of these objects are smaller than 64 byte [10]. The sheer amount of objects and the small data sizes can be found in many other online graph algorithms too. These applications are mostly dominated by read accesses over writes [2]. Whether write accesses are realized as updates or result in a new object version is depending on the system and the type of data.

DXRAM addresses these challenges by keeping all data always in memory. Huge storage capacities can be provided by aggregating many nodes within a data center. A transparent background logging is used to mask node failures and provides persistence even in case of power outages. DXRAM is inspired by many ideas of RAMCloud [12], however instead of larger objects stored in tables we target at managing billions of very small data objects [6]. Furthermore, we do not rely on a single coordinator, but use a super-peer overlay-network for node lookup and recovery coordination of crashed nodes.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 3–15, 2015.
DOI: 10.1007/978-3-319-27308-2_1

In a previous publication we have presented a memory management approach, which allows to store up to one billion small data objects on a commodity node [5]. In this paper we focus on the global meta-data management and the mapping of global IDs to single nodes. The sheer amount of objects raise new challenges on these well studied topics. As RAM is expensive a space-efficient meta-data management with a fast lookup is mandatory.

We propose an approach which merges thousands of global IDs into a single ID-range, resulting in a severely reduced amount of mapping entries. The ID ranges are stored using a modified B-tree structure which allows a fast and space-efficient mapping of billions of global IDs to nodes and a client cache optimized for locality-based access-pattern.

The structure of this paper is as follows. In Sect. 2 we give a brief overview of the DXRAM system including the super-peer overlay which is used to realize the proposed meta-data management. The latter is described in Sect. 3 followed by the evaluation in Sect. 4. Related work is discussed in Sect. 5 followed by the conclusions and an outlook on future work in Sect. 6.

2 DXRAM Architecture

The overall architecture of DXRAM is divided into core and extended services. The core services have been designed to support different data models and data consistency strategies as well as extended services (file system, table-based storage, replication, etc.). One of the main objectives in the core services is to keep the functionality and the interface for high layers as compact as possible. Therefore the core service provides essential functionality for the management, storage and transfer of key-value tuples. The minimal interface for the tuples includes `create`, `delete`, `get`, `put` and `lock`.

2.1 Chunks

A key-value tuple in DXRAM is called *chunk*. A chunk consists of a 64 bit globally unique chunk ID (*CID*) and binary data. Chunks have variable sizes defined during their creation and are always stored en bloc. The operations `get` and `put` work always on full chunks.

The CID is split into two parts. The first 16 bit contain the node ID of the chunk creator (NID_C). The creator of a chunk is not necessarily the actual owner of the chunk. The second part contains a 48 bit locally unique value, called *LID*, which is incremented during each local chunk creation ensuring a sequential ID generation scheme for CIDs on every node. This is also the foundation for a efficient local address translation [6] as well as for the later presented range-based B-tree structure. Furthermore, it allows to support locality-based access patterns.

The CID size is configurable, but with the default setting we can address 65,536 nodes each with up to ≈280 trillion chunks. Furthermore, CIDs of deleted chunks are re-used to keep the ID space compact.

The sequential ID generation is not a constraint for applications. When using databases as persistent storage they typically access data through auto-incremented row IDs similar to our LIDs. For all other applications DXRAM provides a naming service allowing user-defined keys. The intention is that not each single object needs a user-defined key but only a subset, e.g. the user-profile record in a social network referring all other data of a user.

Because RAM is expensive the transparent backup logging replicates chunks on multiple *backup nodes* only in flash memory. By default each chunk is replicated on three backup nodes allowing fast recovery of data in case of a node crash similar to RAMCloud [11].

2.2 Super-Peer Overlay

Around 5–10% (configurable) of the available nodes in DXRAM are dedicated super peers. The super peers form a ring, where the node ID (*NID*) defines the position of the super peer on the ring. The resulting ring has similarities with Chord [16]. However, DXRAM does not use a finger table but the super peers know each other, allowing to contact every node of the ring in $\mathcal{O}(1)$. Because of the high-speed network in a data center and the limited number of super peers this is feasible, in contrast to traditional Internet-scale DHTs. Every super peer manages a range of NIDs between the NID of the predecessor super peer and its own NID. A peer is assigned to the super peer which manages its NID.

Every super peer manages the meta data (existing chunks, assigned backup nodes, etc.) for all its peers. In addition, the super peer also monitors its peers and coordinates the recovery of any failed peer it is responsible for.

Obviously, the location of a chunk given its CID needs to be determined before it can be read or written. The NID part of a given CID is sent with a request to the super peer which is responsible for the corresponding NID range. Each super peer sends a list of all running super peers periodically to its associated peers. If the super peer overlay changes, e.g. a super peer joins or fails, it might happen that a node contacts the wrong super peer. In this rare case the contacted super peer searches for the correct super peer and informs the requesting node.

As mentioned before DXRAM allows the migration of hot spots from one node to another for load balancing reasons. We do not expect millions of object migrations, but we have to be able to resolve potential clustering of hot spots on some nodes. For example in a social network it could happen that some famous artists or pop stars (some artists have up to 40 million likes in Facebook [15]) are stored on one DXRAM node but of course overall there will not be millions of famous stars. During a migration three nodes are involved, the source node, the target node and the super peer responsible for the chunk meta data.

After a node fails the recovered data is stored on potentially many nodes which is handled by DXRAM as multiple migrations.

A peer failure is detected through a heart beat protocol between a super peer and its associated peers or by any other node getting a timeout on a request. If a peer failure is detected the super peer contacts all backup nodes of the failed

node and coordinates the recovery. An explicit backup update order ensures that the recovery can be executed by the backup nodes themselves. In addition the backup order allows the super peer to recover the meta data in parallel during a running node data recovery. For the super peer the recovery of meta data is the same as bulk migrations.

When a super peer fails the stored meta data is lost and the super-peer overlay is broken. Both is widely researched in the context of Chord [16] and no problem. If all meta data replicas are lost or broken it is still possible to reconstruct the missing meta data from peers.

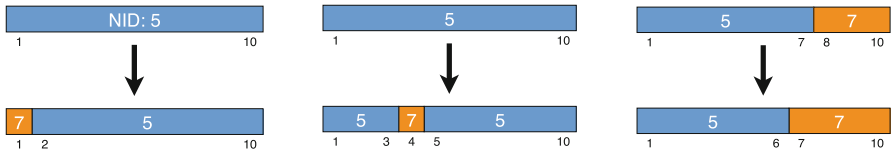
3 CID-Ranges

Many in-memory systems map global IDs to nodes using hash tables allowing lookups in $\mathcal{O}(1)$, e.g. RAMCloud [14] and Trinity [15]. The problem with hash tables is the necessary collision resolution, which gets expensive with increasing load factors. In the worst case the table size needs to be increased and all entries need to be re-hashed. Collision probability can be reduced by larger tables, which however increases the overhead [1]. Another important overhead aspect is that hash tables also need to store the key with the value in each table entry in order to find the correct key in case of collisions. There is no simple way to group multiple entries or reduce entry sizes without losing important information. Overall hash tables are an excellent data structure but we want to minimize the number of meta-data entries.

Databases and table-based storages in general (both disk based or in-memory based) offer column-indices to be created as needed in order to speed up queries. The implementations include many variants of trees mostly based on B- and T-trees as index structures [8]. Balanced trees have lookup times in $\mathcal{O}(log\ n)$ compared to $\mathcal{O}(1)$ for hash tables and can consume considerable memory for pointers and inner nodes. The advantage of ordered tree-based indices is the possibility to scan keys in sequential order (for range queries) and the use of partially filled blocks for fast insertions and deletions. The DXRAM core does not provide key scanning or a table-based data model. However, fast insertions and deletions are necessary for a flexible meta-data management. The inferior lookup time complexity can be compensated through an intelligent client-side caching (see Sect. 3.3).

As mentioned before storing one mapping entry for each chunk is too expensive. If for example one super peer manages 10 peers, each storing one billion chunks, the super peer would have to manage 10 billion mapping entries. Every entry consists of at least a NID (2 bytes) and if necessary a CID (8 bytes), resulting in an overall memory consumption of 20 to 100 GB. All typical data structures like lists, hash tables and indices require additional memory for empty entries, pointers, overhead for the collision resolution, etc.

The sequential CID generation of DXRAM is the foundation for a more compact representation. Instead of storing one entry for each CID, we store one range containing the start and end CID and the NID where all chunks of the



(a) CID 1 is migrated from node 5 to 7.

(b) CID 4 is migrated from node 5 to 7.

(c) CID 7 is migrated from node 5 to 7.

Fig. 1. Effect of chunk migration on CID-ranges.

range reside. In the best case we have to store only one CID range for each node. In the previous example, if there is only one range per node, the memory consumption is reduced by almost 99.99 % to 20 to 100 Byte.

As mentioned before we must support data migrations for load balancing reasons. If a chunk is migrated we have to consider two main scenarios shown in Fig. 1. In both cases the range must be split, resulting in two ranges in (a) and three ranges in (b). In addition a third scenario (c) is shown where two ranges exist and a chunk moves from one range to the another one because of a migration.

We do not expect billions of migrations as there will be only a limited number of hotspots. Thus range splits are rather seldom and are not a space efficiency problem at all. If even 10 million chunks (1 % out of one billion) would be migrated this would result in around 200 million ranges overall, resulting in a memory consumption of 400 to 2000 MB, which is still a space-overhead reduction by over 90 % compared to 20 to 100 GB when using hash tables.

3.1 CID-Tree

Hash- and index-based mappings store one entry for each key, whereas we want to store CID ranges consisting of two keys (the start and end of the range). Such key-key-value (kkv) tuples are rarely researched in the context of in-memory systems and traditional data-structures are not optimized for kkv tuples. We have designed a tree-based data structure for efficiently managing CID ranges, which includes creating, deleting and searching keys in CID ranges.

The CID-Tree approach is based upon a modified B-tree to efficiently manage CID ranges. The start and end LIDs of the ranges are stored inside the inner nodes and the NIDs are the leaves. Figure 2 shows an example of a CID-Tree. A B-tree is divided into ranges in principle. For example, if the tree contains only two entries, there exist three ranges. The first range is between 1 and the first entry, the second range is between the first and second entry and the last range is between the second entry and the maximal value. For every range the tree stores a pointer to a leaf containing the NID of the range.

The migration, deletion and search are executed using (potentially multiple) normal B-tree operations for insert, delete and search and have a complexity of $\mathcal{O}(log\ n)$. During a migration as well as during a deletion it is possible that a key

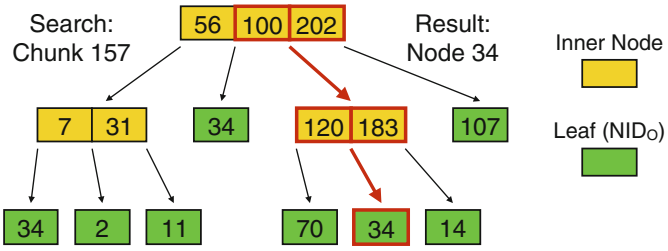


Fig. 2. Searching CID 157 in a CID-Tree returns NID 34

need to be modified, inserted or merged with another key (if a range changes the host node, is split or two ranges need to be merged). A search always requires to follow one path from the root to one leaf of the tree.

The memory consumption of a single entry in the CID-Tree can not be easily calculated, because multiple nodes and pointers have to be considered. The formula below gives a basic approximation:

$$\text{EntrySize} = \frac{2 * \text{Order} * (\Delta\text{Key} + (\Delta\text{Leaf} \vee \Delta\text{ChildRef}))}{\text{EntryCount}}$$

$$\Delta\text{Key} = 6 \text{ byte}, \ \Delta\text{Leaf} = 2 \text{ byte}, \ \Delta\text{ChildRef} = 4 \text{ byte}$$

The entry count of a node is at least the order and two times the order maximum, resulting in 8–20 byte memory consumption for a single entry.

3.2 Backup Nodes Integration

As mentioned before, DXRAM uses by default three backup nodes for every chunk to recover failed nodes. In addition to the actual owner of a chunk (NID_O) the three backup nodes have to be stored as well. Like before we do not store backup nodes for each chunk, but only once for each CID range.

To store the backup nodes together with the NID_O we modify the entry size of the CID-Tree. Instead of a short value (2 byte) for the NID we store a long value (8 byte). This increases the size of an entry by additional 6 byte. In the lowest 2 byte of the long value the NID_O for the range is stored. In the other 6 byte the NIDs of the three backup nodes are stored (NID_{B3}, NID_{B2}, NID_{B1}, NID_O). Backups are processed in their order during fault-free execution and a simple shifting to the right repairs the meta data during recovery.

In order to allow a parallel and fast recovery the chunks of a node have to be distributed across many backup nodes [11]. We achieve this by limiting the size of one CID range entry, e.g. 2 million chunks in one CID range.

In the previously used example with 10 billion mapping entries, the integration of the backup nodes results in 5,000 CID ranges (instead of only one) per node, consuming around 32 to 64 KB of memory which is still very space efficient.

3.3 Client-Side Caching

The presented CID-Tree is not only used on super peers, but can also be used without modification on the peers for caching. Super peers answer a node lookup request not only with a single NID, but with the corresponding range (with the NID and the start and end LID), in which the requested CID is located. The requesting peers cache CID ranges in a local CID-Tree and requests to a super peers are only sent in case of a cache miss. Due to space limitations we cannot discuss cache eviction strategies here.

4 Evaluation

In this section we present and discuss the evaluations of the proposed data structures and the client-side caching. We also compare the overall DXRAM performance with MongoDB and Cassandra using the social network benchmark BG.

The data structure evaluation is performed on a single super peer (Intel® Core™ i5-2400 CPU, 4 cores, 3.1 GHz, 32 GB RAM) using Ubuntu 11.04 and the OpenJDK Java Runtime Environment (version 7). The other evaluations are performed on a private cluster using nodes connected by a Gigabit Ethernet network. Each node is equipped with an Intel® Xeon® CPU E3-1220 V2, 4 cores, 3.1 GHz, 16 GB RAM (DDR3) and a 1 TB local disk (HUA722010CLA330). All nodes are running Debian 7.6 and the OpenJDK Java Runtime Environment (version 7). Java is required by the BG Benchmark, DXRAM and Cassandra.

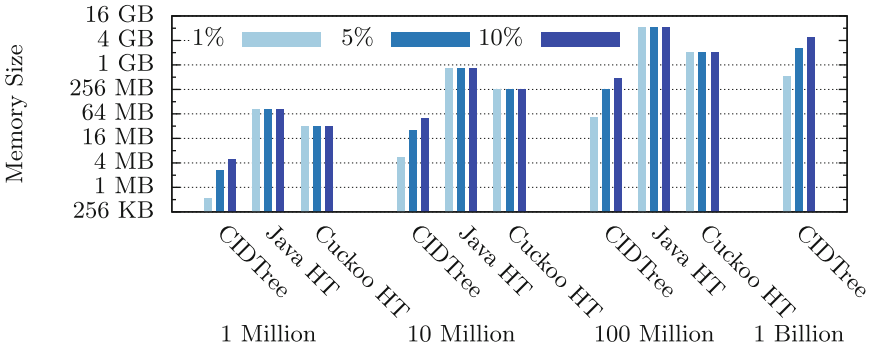
4.1 CID-Tree

The first evaluation compares the proposed CID-Tree with the Java built-in hash table and a cuckoo hash table using primitive data types on a single node.

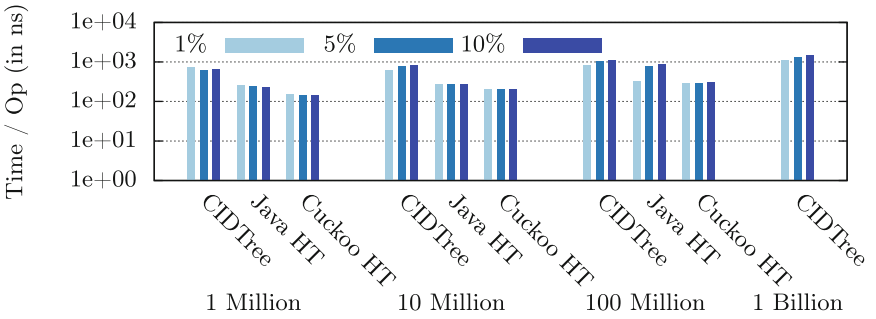
Every data structure is filled with 1 million, 10 million, 100 million and 1 billion meta-data entries and afterwards 1 %, 5 % and 10 % of the entries are randomly migrated. Afterwards, we measured the memory size of the data structures and the time per migration and per get operation. The throughput of operations is only measured on the localhost to get a better insight of the raw performance of the data structures. The evaluations are repeated at least 10 times with only minimal variation of the results.

Figure 3 shows the results of these measurements. The migration times are very fast for both hash table solutions. The CID-Tree is noticeably slower for 1 million objects but its performance improves for larger object sets, where it is only around 3 to 5 times slower than the hash tables.

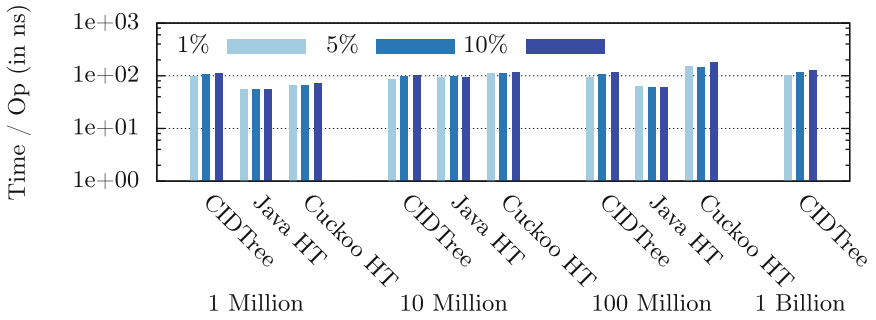
The get operations are fast when using Java hash table. The CID-Tree is here slower by a factor of 2. The cuckoo hash table has fast access times for 1 million entries, but gets slower when more entries have to be stored. Although the proposed CID structure has access times in $\mathcal{O}(log\ n)$ compared to $\mathcal{O}(1)$ for the hash tables, the evaluation shows that the access times are only slowly growing and are nearly constant for a very large number of entries.



(a) Memory size of the data structures.



(b) Time (in ns) per migrate operation.



(c) Time (in ns) per get operation.

Fig. 3. Comparison of the CID-Tree, the Java hash table and a cuckoo hash table.

It can be observed that the memory size of the CID-Tree is very low. The size of the cuckoo hash table is around 50 times greater and the Java hash table size is even around 150 times greater. The experiment with one billion entries could not be performed for the hash tables because the physical node run out of memory.

The evaluation results show that for managing large-scale object sets the proposed CID-Tree is the best approach. We trade a small amount of speed for a much better space efficiency but still all operations of CID-Trees are much faster than the network request roundtrip times of clients adding only 1 μs or less when managing one billion entries.

4.2 Client-Side Caching

In this evaluation DXRAM is set up on three cluster nodes (configuration see above). The first node acts as super peer using a CID-Tree for the meta-data management. The second node is a storage peer managing chunks and requesting migrations of CID ranges (length 1 to 100 CIDs). It is likely that in many cases not single small chunks are migrated but a set of chunks, e.g. the profile of a famous person in a social network together with its friend list, photos, comments etc. (although not all these chunks will be stored within one CID range at least some of them will be, e.g. the friendship list). For the experiments we migrated 1 %, 5 % and 10 % of the chunks on the storage node. The third node emulates a user requesting all CIDs of the chunks created before. This node also runs the client-side cache (if enabled) based on the CID-Tree as well.

The evaluation (with multiple evaluation runs) which can be seen in Fig. 4 shows that the client-side caching works very well and can greatly reduce the access times by up to 98 %.

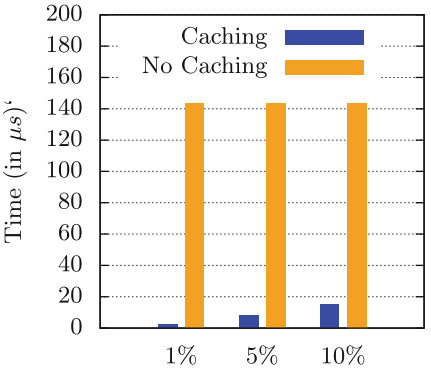


Fig. 4. Time per operation with enabled and disabled client-side caching.

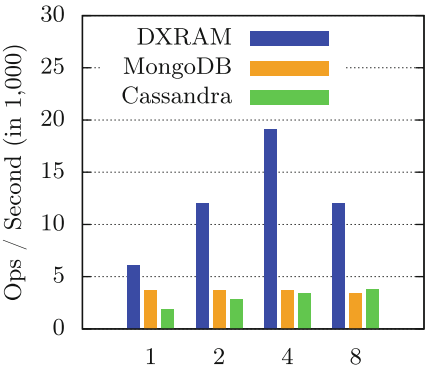


Fig. 5. BG-Benchmark evaluation of DXRAM, MongoDB and Cassandra.

4.3 BG Benchmark

BG is an open-source benchmark to evaluate the performance of data storages for interactive social media networks [3]. It supports a variety of state-of-the-art

storage systems like for example MongoDB [13] and Cassandra [7]. BG emulates interactive social networking actions known from Facebook, Twitter and YouTube (more details see [3]).

In the following experiment we compared DXRAM with MongoDB (2.6.4) and Cassandra (3.0.0 snapshot) using BG. We developed a BG client for DXRAM and for MongoDB and Cassandra we used the ones provided with BG. BG was configured to create 10,000 profiles, each with 100 friends and resources resulting in a graph of over 1 million nodes and 2 million edges. The requests are performed with 4 BG nodes executing up to 512 threads (each thread simulating one client). The number of used storage nodes range from 1 to 8 nodes for each system. In the given scenario DXRAM is started with one super peer and MongoDB using sharding, one config server and one query router.

Figure 5 shows the result of the evaluation. MongoDB performs around 3,500 actions per second in every test run limited by the query router, which is the bottleneck of MongoDB. DXRAM scales very well for 1, 2 and 4 storage nodes with results of 6,000, 12,000 and 19,000 actions per second. The performance with 8 storage nodes is lower because the clients could not create enough workload to fully load the storage nodes. For Cassandra we used the default partitioner *Murmur3Partitioner* which distributes rows of tables across storage nodes using consistent hashing for the primary key of a table row. In principle this allows a good load balancing, however for BG we expect a lot of cross-node traffic as this does not support locality of data. For example the friend list entries of a profile might be scattered over several nodes and thus when requesting all friends of a user will cause a lot of network traffic, which explains the lower performance for BG. Nevertheless, we can still see the scalability of Cassandra as throughput increases from 1,800 actions per second to 3,700 when the number of storage nodes is increased as well as the number of clients performing actions.

We also tested DXRAM with up to 500,000 profiles using 4 storage nodes. The resulting graph contains up to more than 50 million nodes and 100 million edges. With 8 BG nodes we achieved a throughput of around 17,000 to 22,000 operations per second. Due to problems with BG we could not perform these tests with MongoDB and Cassandra.

5 Related Work

Meta-data management has been and will be an important research area where much results have been published somehow related to our work. Because of space constraints we can only discuss the most relevant work.

RAMCloud is a distributed in-memory system organizing data in tables [12]. Both memory and backup disks use the same log-structured design. The mapping of 64 bit global IDs to nodes is implemented by a hash table maintained by a central coordinator. The central coordinator is not necessarily a single node, but normally 3 to 7 servers using a consensus protocol [14]. Overall, we are inspired by many ideas from RAMCloud but we aim at supporting many small objects (key-value tuples) whereas RAMCloud does focus on a table-based data model for larger data items.

Trinity Graph Engine is a distributed in-memory graph-database, designed to support algorithms executing on graphs with billions of nodes [15]. It uses a key-value data model implemented in C# on top of a memory cloud. Node lookups use a hash function to first find a memory partition (trunk). The node storing a trunk is found through a globally shared address table. Finally, on this node another hash function is used to get the offset and size of the key-value pair inside the trunk. Because Trinity is not available as open source we could not compare it with our approaches. However, from the evaluations we made, we believe that hashing is less space-efficient than our approach, which allow us to manage even larger graphs in the future.

FaRM utilizes the memory of machines in a cluster as shared address space to implement a distributed computing platform using RDMA for network communication [4]. For low latency network access on remote nodes the operating systems and the NIC device drivers are modified. A hopscotch hashing approach combined with chaining is used for meta-data management allowing a lookup in a small number of RDMA reads. On the contrary DXRAM uses a B-tree structure with only a single network request (instead of multiple RDMA reads) and allows the caching of lookup responses. In Addition the objects stored in FaRM are slightly larger (64 Byte to multiple MB) than in DXRAM.

MongoDB is a NoSQL database and structures its data in documents similar to JSON objects [13]. Documents contain various fields including primitive data types, other documents and array of documents. MongoDB caches as much data as possible in memory and swaps data to disk, if another process needs memory.

Cassandra is a NoSQL database for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure [7]. Cassandra nodes use a DHT for meta-data management and virtual nodes to evenly distribute keys onto available peers. Obviously, Cassandra as well as MongoDB and other NoSQL databases are designed to run on disks or SSDs providing tables, indices and a SQL-like query language whereas DXRAM is designed to keep all data always in RAM with a focus on very small objects.

BlobSeer is a storage system for large unstructured binary data (blobs) up to 1 TB [9]. The blobs are split into blocks of fixed size (data striping) and distributed among storage providers. Every block is referenced as a range (offset, size) of the blob and the corresponding meta data is managed in a distributed segment tree. On every level of the segment tree the range of the father is halved to the left and right son. In contrast, DXRAM targets many very small objects and CID-ranges contain the meta data of potentially many objects whereas a range in BlobSeer contains a single Block of a large object. The B-tree in DXRAM allows very flexible range sizes contrary to the rigid range size in a segment tree. Finally, tree structures are commonly used in *distributed file systems* for storing locations of file blocks or file block ranges. In contrast DXRAM merges IDs of multiple different objects (not blocks of a single) to ranges.

6 Conclusions

Large-scale interactive applications and online graph analytics often need to process billions of small data objects in short time, which makes fast networks and in-memory storage mandatory. A fast and space-efficient meta-data management is very important for such systems. In this paper we have proposed a range-based meta-data management combined with a super-peer overlay. Each super peer manages tree-based data structures mapping global chunk IDs to peers based on ID ranges. The evaluation shows that the proposed approach has a much better space efficiency than hash tables while at the same time being not much slower ($1\,\mu s$ or less for one billion entries). The overall system performance shows a very good throughput for a social network benchmark, compared to state of the art NoSQL databases. We plan to do more evaluations (including graph algorithms) with more nodes and more data.

References

1. Askitis, N.: Fast and compact hash tables for integer keys. In: Proceedings of the Thirty-Second Australasian Conference on Computer Science, ACSC 2009, Darlinghurst, Australia, vol. 91 (2009)
2. Atikoglu, B., et al.: Workload analysis of a large-scale key-value store. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2012, New York, NY, USA (2012)
3. Barahmand, S., Ghandeharizadeh, S.: Bg: a benchmark to evaluate interactive social networking actions. In: CIDR. Citeseer (2013)
4. Dragojević, A., Narayanan, D., Castro, M., Hodson, O.: Farm: fast remote memory. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014), pp. 401–414. USENIX Association, Seattle, April 2014
5. Klein, F., Beineke, K., Schöttner, M.: Memory management for billions of small objects in a distributed in-memory storage. In: IEEE Cluster 2014, September 2014
6. Klein, F., Schöttner, M.: Dxram: a persistent in-memory storage for billions of small objects. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2013), pp. 103–110, December 2013
7. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
8. Lu, H., Ng, Y.Y., Tian, Z.: T-tree or b-tree: main memory database index structure revisited. In: 11th Australasian Proceedings of Database Conference, ADC 2000, pp. 65–73 (2000)
9. Nicolae, B., Antoniu, G., Bougé, L.: Enabling high data throughput in desktop grids through decentralized data and metadata management: the blobseer approach. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 404–416. Springer, Heidelberg (2009)
10. Nishtala, R., et al.: Scaling memcache at facebook. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013), Lombard, Illinois (2013)
11. Ongaro, D., et al.: Fast crash recovery in ramcloud. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, New York, NY, USA (2011)

12. Ousterhout, J., et al.: The case for ramclouds: scalable high-performance storage entirely in dram. SIGOPS Oper. Syst. Rev. **43**(4), 92–105 (2010)
13. Plugge, E., Hawkins, T., Membrey, P.: The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing, 1st edn. Apress, Berkely (2010)
14. Rumble, S.M.: Memory and Object Management in RAMCloud. Ph.D. thesis, Stanford University (2014)
15. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA (2013)
16. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM 2001, pp. 149–160. ACM, New York (2001)