MaiterStore: A Hot-aware, High-Performance Key-Value Store for Graph Processing *

Dong Chang, Yanfeng Zhang, and Ge Yu

Northeastern University, Liaoning, 110819, China
cdaspirin@gmail.com,zhangyf@cc.neu.edu.cn,yuge@mail.neu.edu.cn

Abstract. Recently, many cloud-based graph computation frameworks are proposed, such as Pregel, GraphLab and Maiter. Most of them exploit the in-memory storage to obtain fast random access which is required for many graph computation. However, the exponential growth in the scale of large graphs and the limitation of the capacity of main memory pose great challenges to these systems on their scalability.
In this work, we present a high-performance key-value storage system, called MaiterStore, which addresses the scalability challenge by using solid state drives (SSDs). We treat SSDs as an extension of memory and optimize the data structures for fast query of the large graphs on SSDs. Furthermore, observing that hot-spot property and skewed power-law degree distribution are widely existed in real graphs, we propose a hot-aware caching (HAC) policy to effectively manage the hot vertices (frequently accessed vertices). HAC can conduce to the substantial acceleration of the graph iterative execution. We evaluate MaiterStore through extensive experiments on real large graphs and validate the high performance of our system as the graph storage.

Keywords: graph store; key-value store; hot-aware cache; SSDs; Maiter

1 Introduction

In the era of big data, a huge amount of graph data is being collected, e.g., Twitter follow graph, Amazon consumer-purchase-record bipartite graph, LinkedIn social graph etc. A lot of state-of-the-art graph computation frameworks [21, 9, 27, 29, 18, 17, 20, 19] are proposed to process and mine these big real-world graphs. In many graph algorithms, the computation of a graph vertex only depends on its neighbors' values. It could involve a substantial amount of non-contiguous access when accessing values of the neighbors. Therefore, most of these graph computation frameworks [21, 9, 18, 19] are designed to maintain the graph data in memory for fast random access.

* This work was partially supported by National Natural Science Foundation of China (61300023, 61272179), Fundamental Research Funds for Central Universities (N120416001, N120816001), China Mobil Labs Fund (MCM20122051), and MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06).

However, real-world graphs might be too large to fit into memory. Even though high-end servers with high-capacity memory are available in recent years, it may not be a typical solution for the widespread adoption of memory storage. The steep price and relatively small capacity of DRAM will continue to be a concern for a long time. On the other hand, many big data processing systems [1, 2] and several graph computation systems [29, 20] utilize external storage devices such as HDDs (hard disk drivers) to store these large graphs, but the costly I/Os correspondingly render these systems' performance inefficient. Thus, in graph computation frameworks, we believe that the graph-structured data should not be simply stored in the disk-based storage or the memory storage, but instead be placed in the fittest storage position while exploiting both advantages of disk and memory.

To address this challenge, we introduce **MaiterStore** for storing large graphs by using the state-of-the-art solid state drives (SSDs). We observe that many graphs are stored in *in-memory key-value tables*, such as Maiter [21]. The states of the graph vertices/edges are required to update many times during the computation. On the other hand, the graph structure, e.g. adjacency list, is immutable during the whole iterative computation. In addition, the size of graph-structured data is usually far larger than the state data of graph. MaiterStore keeps the frequently updated volatile graph state data in main memory for fast random access, while store the read-only immutable graph-structured data on SSDs. This is mainly because SSDs are capable of gaining better I/O performance than the HDDs, and are cheaper than DRAM. Using SSDs to store the immutable graph-structured data will bypass the I/O bottleneck of HDDs.

MaiterStore equips the graph data with key-value pairs and has an efficient *page manager* to manage these key-value pairs on SSDs. In addition, considering the graph data access pattern, MaiterStore adopts a page-based *prefetching buffer* in memory, which prefetches the anticipating graph-structured data from SSDs. The prefetching buffer can adapt to the access pattern changes and exchange the prefetched SSDs pages dynamically. Specifically, the graph computation usually exhibits skewed access patterns where some vertices' edges are accessed frequently and excessively. Based on this observation, we design a *hot-aware cache (HAC)* in memory to cache the hot vertices and their edges from SSDs intelligently. HAC can conduce to the substantial acceleration of the graph processing.

We summarize our main contributions as follows:

- MaiterStore, a high-performance key-value store for large-scale graph processing.
- HAC, a hot-aware cache for caching the hottest part of immutable graph-structured data during the graph computation.
- Implementation of MaiterStore and evaluation of its performance with graph applications on a local cluster as well as Amazon EC2.

The rest of this paper is organized as follows. Section 2 introduces the graph computation framework—Maiter and SSDs characteristics. Then we present the system design of MaiterStore, and its components as well as APIs in Section 3.

The experimental results are shown in Section 4. We outline the related work in Section 5 and conclude the paper in Section 6.

2 Preliminaries

2.1 Introduction to Maiter

In this paper, we will present MaiterStore to support a recently proposed in-memory graph processing framework Maiter [21]. Note that, although Maiter-Store is currently designed to make Maiter memory-efficient, the techniques used are also suitable for other graph processing systems if they maintain graph data based on key-value tables. The APIs can be easily extended and be compatible with other systems.

Maiter is built on the basis of a novel computation model, called delta-based accumulative iterative computation. By exploiting this new model, Maiter shows better performance than many of the state-of-the-art graph processing systems. Maiter follows a master-slave architecture for distributed computation. For graph processing, Maiter firstly splits the input graph data into multiple partitions and sends to different workers.

On each worker, the received graph partition is loaded in an in-memory key-value store, *state table*, as shown in Fig.1. The first field represents the id of a vertex; the 2nd, 3rd, 4th fields correspond to the vertex state, which are updated during computation; the last one is vertex adjacency list, which is always immutable and static but frequently accessed during the iteration. We will explore this mutable/immutable property of graph data to design our graph storage. In addition, Maiter exploits a *priority scheduling* policy, which identifies the key vertices during graph processing and assigns them higher execution priority. Consequently, these vertices with higher priority are the hot-spot of the graph. We will design a caching mechanism to identify and cache these hot vertices for fast access.

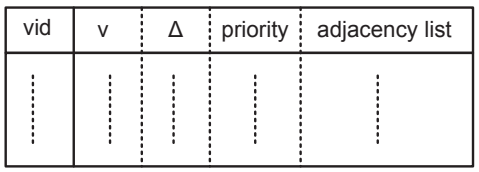

Fig. 1. In-memory state table in Maiter

2.2 SSD Characteristics

SSDs have the same purpose as conventional mechanical hard drives, but there is one crucial difference: they are electronic devices without any mechanical

moving parts. Unlike HDDs, SSDs do not store data on spinning platters, but use flash memory instead. Therefore, SSDs are capable of providing *fast random access speed*, and the time to access data is almost proportional to the amount of data irrelevant for the physical locations of data in SSDs. As opposed to disk, accessing two sequential pages is no faster than accessing two random pages.

The basic I/O unit of SSDs is a *page*, typically 4KB in size. These pages are organized into blocks which are between 256KB or 1MB in size. However, when data stored in the SSDs will be updated in place, SSDs have to perform an erase operation which cannot erase selectively on a specific data record, but on an entire block containing the data record. This leads to redundant effort and further slows the speed of update operation. MaiterStore is a hybrid storage system using SSDs. Considering the *erase-before-write* [11] limitation, MaiterStore is designed to bypass this limitation and optimize SSDs I/Os.

3 MaiterStore Design

In this section, we present the system architecture of MaiterStore and the related techniques. The design of MaiterStore is driven by the Maiter's memory-inefficient state table.

In Maiter, gigantic graph data exhausts the limited memory of each worker. In order to get rid of the bottleneck caused by limited memory, we ought to *destage the huge static graph-structured data to SSDs*, reducing pressure on the memory. After migration, the state table in Maiter could become relatively smaller than before. Meanwhile, using SSDs could not lead to a sharp slowdown of random access compared to the memory. From the fact that every vertex in graph storage is an index to its neighboring vertex, thus we regard the vertex and its adjacency list as static key-value pair storing the SSDs.

3.1 System Overview

Overall, MaiterStore is composed of the following key components, as shown in Figure 2.

Page Manager: The page manager serves to equip the static graph-structured data as key-value pairs and store them on SSDs in units of page. Meanwhile, it selects the first key for each page as *page number* and constructs these page numbers as the *BST index*. We will describe this in Section 3.2.

Prefetching Buffer: The prefetching buffer is a page-based storage structure that is maintained in memory. It is used for fetching the to-be-accessed pages from SSDs early enough so that they are available in the buffer when required. More details will be described in Section 3.3.

Hot-aware Cache (HAC): This is a fixed-size read cache of key-value entries that is maintained in the memory. We use a novel replacement strategy to evict the key-values pairs when inserting items into the full HAC. We will present this in Section 3.4.

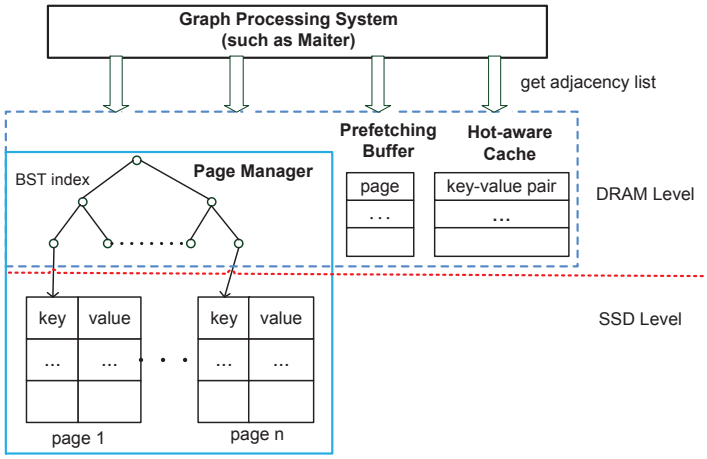

Fig. 2. MaiterStore architecture on each worker

For better understanding how these components work together, we describe a simple example, a lookup operation of the adjacency list.

In this example, we divide the graph of six vertices into three equal pages. These vertices are flash resident and paged in and out of the memory as needed. Assume also that prefetching buffer and HAC are empty from the beginning. When graph processing systems, such as Maiter, compute the PageRank of vertex 1, requiring to query the adjacency list `L` for vertex 1, the HAC is consulted first. If `L` is in the HAC, MaiterStore returns the cached adjacency list. If `L` is not in the HAC but it is in the prefetching buffer, it is read into the HAC from the prefetching buffer. If `L` is neither in the HAC nor in the prefetching buffer, the page manager firstly finds the page number `page0` via the BST index to locate the position of the adjacency list on SSDs, and then fetches the corresponding page into the prefetching buffer and inserts the adjacency list into the HAC. As shown in Figure 3. If the HAC is full when a new adjacency list is read in, the HAC must evict an item according to its replacement policy. In the following, we will outline the three components in detail.

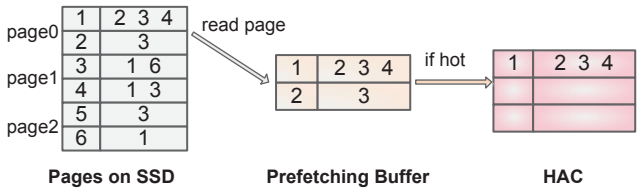

Fig. 3. Example of the lookup operation on MaiterStore

3.2 Page Manager

Before iterative computation, Maiter splits the input graph data into multiple partitions and sends to different machines. For each worker machine, the page manager is responsible for decomposing the graph partition into contiguous equal-size pages. Then these pages are sequentially logged in SSDs. The log-structured manner and its benefits have been reported in earlier work [30, 15, 12]. Typically, a page is a collection of graph vertices and their outgoing edges; in other word, edges are indexed for their source vertices. For each page, we choose the key of first pair as *page number* and the page manager saves it on the local machine. Every key-value entry in the page is associated with this page number. Before iterative execution, the page manager constructs these page numbers a BST (balanced search tree) index.

BST index is a simple but efficient in-memory index. We can quickly locate the page by binary search in logarithmic time. Also note that the BST index is a sparse and memory-efficient data structure since we only maintain the first key of the page.

Since the key-value pairs on SSDs are maintained implicitly in a sorted order, we can easily obtain the page number via the BST index and locate the retrieved page. For example, the request of a key-value pair (k, v) is sent to the page manager, the page manager can readily map the key k into the page number via BST index. And then, the page manager can quickly find the corresponding page on SSDs according to the page number.

The page in our key-value system is the basic unit that can be read or written. Generally, the page size must be an integral multiple of the SSDs default page size 4KB to keep I/O aligned. The page size is set as 8KB by default in MaiterStore. We will further study the effect of page size on performance in Section 4.2.3.

For page management, the page manager adopts a *write-once-read-many* [7] model. This model could not only enable high throughput of data access but also potentially avoid SSDs write pitfalls.

3.3 Prefetching Buffer

Prefetching is aimed at making data available in the memory before it is requested, thereby hiding the effect of latency resulting from poor reference locality. However, prefetching is not cost-free and it has to manage the prefetched data. If the prefetched data is not subsequently used, it obviously reduces the efficiency of system. To maximize the performance of system, the prefetching technique needs to predict the access pattern, minimizing the number of useless prefetches.

In Maiter, all local vertices stored in state table are processed iteratively round by round. In each iteration, the vertices are processed in the order that they are stored in the state table. Further, these vertices' edges are stored in SSD pages. We may pin a set of pages to memory for prefetching buffer.

In MaiterStore implementation, we adopt the following prefetching method: when the read of the key-value pair is not present in memory, the page manager

requests the page containing the key-value entry and several pages adjacent to that page with multi-threads in advance. In general, the number of prefetching pages and prefetching buffer size are application configurable by users in MaiterStore.

The prefetching buffer can be managed in many way. Finding the locality of graphs is still a big challenge [31], so the prefetching buffer has no information about which page will be reused if it remains in the prefetching buffer. For simplicity, we use FIFO (First In First Out) policy to evict the pages when inserting new pages into full prefetching buffer in current implementation.

In our experiment, it is also shown that prefetching technique can reduce excessive I/Os efficiently and hide its latency effect.

3.4 Hot-aware Caching Policy

Typically, workloads of many graph computation systems exhibit excessive access skew. This is mainly because most real-world graphs obey the power law distribution [28]. For example, the social network can naturally be abstracted as a graph, where pages correspond to vertices and hyperlinks correspond to directed edges. In such real-world graph, most vertices have a relatively small degree but some outliers, such as celebrities in a social network, are much larger. This is important, because computation converges slower on these important vertices than others, and it is desirable to focus computation on them. For graph processing, the vertices with high degree are hot and accessed frequently, but others with small degree are cold and accessed infrequently. Obviously, it would make sense to reside such hot and high-degree vertices in memory. Cold vertices should be migrated to external memory such as flash to mitigate the memory pressure.

Additionally, updating a part of the important vertices selectively [20] rather than entire vertices can lead to more efficient graph computation, because not every element in the intermediate result changes at each iteration. To this end, we should focus on the hot vertices and keep such performance-critical vertices in memory.

HAC attempts to take into account the hotness of graph vertices based on the priority-based iterative execution of Maiter [21], which could avoid loading inactive vertices from SSDs. The hotness of the vertex is normally set to *the priority field* in Maiter state table. In the graph computation framework, Zhang et al. [20] proved priority scheduling: given an execution priority to vertices can potentially accelerate the convergence, because some of vertices play a decisive role in determining the final converged outcome. Under these conditions, we should keep these decisive vertices in cache. Our hot-aware cache occupies small available portions of memory, but holds the top hot key-value pairs and does not incur high penalties to main memory. Fig.4(a) shows the hot-aware cache with the triple(hotness, vid, adjacency list).

Conventional LRU (Least Recently Used) replacement strategy evicts the objects in cache that have not been accessed for the long time, and it only considers the time of the last access to objects. Unlike LRU, HAC can perform

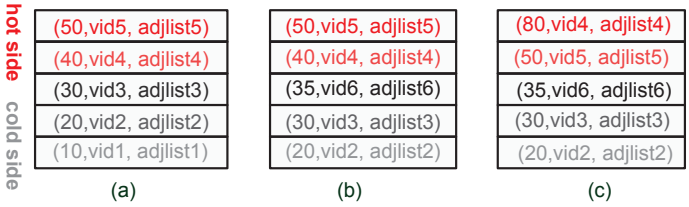

Fig. 4. (a) hot-aware cache. (b) evict the pair with lowest hotness and insert new pair with higher hotness (35,vid6,adjlist6) when miss. (c) update the hotness of the pair (80,vid4,adjlist4) when hit

hot-aware policy to cache key-value pairs with higher hotness. Therefore, the relatively small memory can cache more valuable key-value pairs, which can make a big difference when Maiter executes iterative computation.

Looking-up operation with a key and a hotness value h in MaiterStore needs to read the cache. Upon a miss, the current key-value pair read from prefetching buffer or SSDs will be inserted into cache after evicting another key-value pair(if cache is already full). To decide which key-value pair to evict when cache is full, HAC firstly finds the key-value pair whose hotness is lowest, say h_{lowest}. If $h > h_{lowest}$, HAC evicts the key-value pair with lowest hotness to make room for the current key-value pair in Fig.4(b). If the key-value pair is already in cache, HAC needs to only find the pair and update the hotness as shown Fig.4(c). With the in-memory hash table, HAC achieves O(1) time for hit and eviction.

3.5 API in MaiterStore

MaiterStore is implemented in circa 1800 lines of C++ code using boost library. It currently provides Maiter with graph storage, though it could later be expanded to support other in-memory graph processing frameworks. Basic operations in MaiterStore are as follows:

```
virtual void parseKV(string line, K* vid, V* adjList) = 0;
V get(const K vid);
void put(const K vid, const V adjList);
```

where K and V are the type of key and value respectively. The interface *parseKV* is a user-defined function for converting string record to users' data structure and implemented by developers. The function *get* and *put* are two basic operations in MaiterStore. Both of them are invoked by the upper graph processing frameworks when obtaining or storing graph adjacency lists.

4 Performance Evaluation

In this section, we evaluate our system with extensive experiments. MaiterStore is equipped to Maiter to show the performance in the context of two typical applications PageRank and Connected Components.

4.1 Environment Setting

The experiments are conducted on a cluster of local machines consisting of 4 commodity machines, as well as on Amazon EC2 Cloud [3]. Each machine in the local cluster has Intel Core i3-2120 3.3GHz CPU, 3GB of RAM, Seagate Barracuda 500GB hard drive and Samsung 840 Series SSD [6]. It is running Ubuntu 13.04 with the Linux kernel 3.8.8 and Ext4 file system with default configuration. The Amazon EC2 cluster involves 30 m3.xlarge nodes. Each node has 15GB of memory, and 80GB of SSD.

We evaluate MaiterStore using three datasets: Web-Google [5], Web-BerkStan [5] and Web Graph [4]. The dataset statistics are presented in Table 1.

Table 1. Statistics of datasets

Dataset	Nodes	Edges
Web-Google(55MB)	916,428	6,078,254
Web-BerkStan(62MB)	685,230	7,600,595
Web Graph(11GB)	50,000,000	686,231,717

4.2 Experimental Results

In this section, we present our experimental results. Without loss of generality, we fetch a page into the prefetching buffer and set buffer size as 2 page.

4.2.1 Performance of MaiterStore

To study the effects of different techniques used in MaiterStore, we evaluate MaiterStore with different configurations on the local cluster. For comparison, we also configure Maiter to store graph data in memory only and in disk file only. There are totally seven settings considered for comparison: (1) in memory only(S1); (2) in SSD + Prefetching Buffer + HAC, i.e., MaiterStore(S2); (3) in HDD + Prefetching Buffer + HAC(S3); (4) in SSD + Prefetching Buffer(S4); (5) in HDD + Prefetching Buffer(S5); (6)in SSD only; (7)in HDD only. Figure 5 shows the running time of different configurations on PageRank and connected components. Due to long running time in the final two settings, we do not plot them in Fig.5. In this experiment, we use Web-Google dataset. By comparing S1 with S2 or S3 in Fig.5, although S1 outperforms S2 or S3, our dataset is

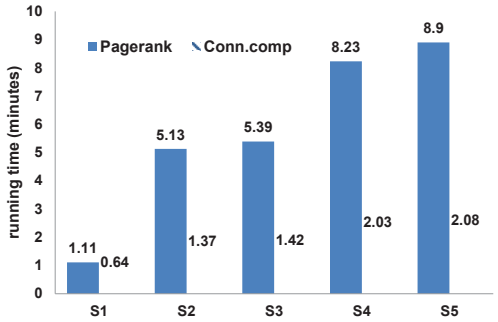

Fig. 5. Comparison of running time under different settings.

much smaller than the capacity of memory. S1 is not scalable for processing huge amounts of the graph data in real applications with limited memory.

Interestingly S2 and S3 have the same running time approximately. This is not surprising, since MaiterStore is not read-intensive for SSD or HDD and most of read operations are cached in the prefetching buffer or the HAC, which causes the read operations concentrating on the hot vertices and hides the high I/O cost. In section 4.2.4, we show that the reads for SSD or HDD are rather few in number. In addition, in such setting most or all of the hot vertices of dataset fit into the HAC, and there is no need to read data from SSD or HDD.

S4 or S5, by contrast, only caches the neighboring vertices and may not cache the hot vertices, and this causes miss ratio increased. We will study the effect of HAC further in section 4.2.4.

In SSD or HDD only settings, we first locate the page and read it from the SSD or HDD, and then find the corresponding key-value without caching and buffering. In PageRank, the running time in SSD only setting is more than 13 times greater than MaiterStore (SSD + Prefetching Buffer + HAC) and the running time in HDD only setting is more than 17 times. This also shows the performance of MaiterStore is affected by the prefetching buffer and hot-aware cache closely.

Based on the above experiments, we can validate that, although the performance of MaiterStore has a certain gap compared with in-memory only, MaiterStore can process much bigger graphs, as it is not limited by the capacity of DRAM. It also shows that MaiterStore performs well on both SSD and HDD.

4.2.2 Scalability of MaiterStore

To show the scalability of MaiterStore under large-scale distributed environment, we conduct connected components on dataset Web Graph using EC2 cluster. Figure 6 shows the running time of MaiterStore when the number of workers is varied. We can see that, the drop from 23.32 to 6.86 minutes using 6 times as many workers represents a speedup of about 4. It demonstrates that for MaiterStore the runtime decreases linearly in the worker size.

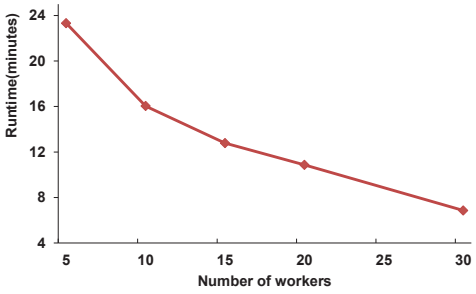

Fig. 6. Connected Components: varying number of workers on EC2 cluster

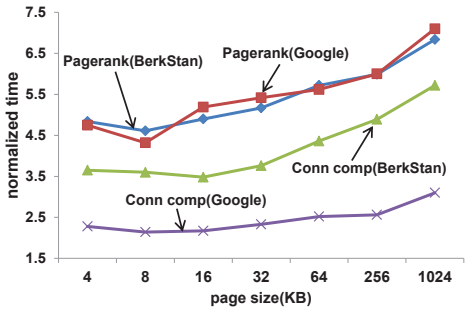

Fig. 7. Effect of page size on performance

4.2.3 Effect of the Page Size

Page size is a vital parameter in MaiterStore. The setting of page size determines the granularity of buffer unit, which has effect on prefetching advantage. A large page size is desirable for reducing index size. On the other hand, a small page size can effectively eliminate reading unnecessary key-value pairs, since the page may contain both hot and cold key-value pairs. To specify the page size, the page size must be an integral multiple of the 4KB, the default page size of the SSD.

We compare performance of different page sizes in stand-alone environment, including the Web-Google dataset and Web-BerkStan dataset in Fig.7. We only cache 10% vertices in HAC. We get the results with various page sizes and show the execution time normalized to that of running on the memory-only setting. We can see that with the large page size, the running time increases significantly, because both hot and cold vertices may co-exist in a page, whereas sometimes we need the hot data only, which causes the unnecessary reads and prolongs the runtime. For the small page size (4KB), the running time is larger than that with page size (8KB). This is mainly because sometimes Maiter requires to compute a

wide set of vertices on different pages, whereas small page contains less vertices than large page, thus causing the miss rate to increase.

The experiment also illustrates that choosing a proper page size can optimize performance. In MaiterStore, we use a page size of 8KB in default.

4.2.4 Comparison with the HAC VS. FIFO VS. LRU

In this section, we compare the effect of different caching techniques in stand-alone environment using the Web-Google dataset. LRU and FIFO could not cache any vertices without any settings, because they are prone to the recently accessed objects. In the experiment, we filter the vertices with smaller hotness in FIFO and LRU. That is, if the hotness of vertex exceeds a predefined threshold, we cache such the vertex. However, in HAC, we have no such setting.

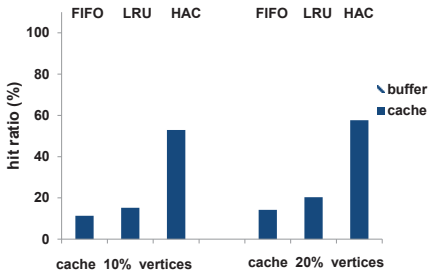

Fig. 8. Effect of cache size under HAC, FIFO and LRU

Fig.8 shows the experimental results while varying the cache size. Even though all the cache policies exhibit better performance as the cache size increases, the HAC is more sensitive to the size of cache than the others. On the other hand, the hit ratio of HAC is more than threefold as high as the two other cache policies. In addition, the hit ratio of buffer is 97.2% when cache is zero, and when cache has 10% vertices size, the total hit ratio exceeds 99.97%. It shows that below 0.03% retrieving data is obtained by accessing SSDs.

4.2.5 Effect of the HAC and Prefetching Buffer

Finally, we evaluate the effect of buffer and cache in stand-alone environment using the Web-Google dataset. The results are shown in Fig.9. The X axis shows the time interval of execution time, and the Y axis shows the hit times during the corresponding time interval. As iteration goes on, the hit times are increased significantly and our HAC plays an increasing important role. The hit times in the last time interval are declined because the execution is terminated during this time interval and corresponding the running time is less than 60s. It shows HAC is gradually caching the hot and performance-critical vertices.

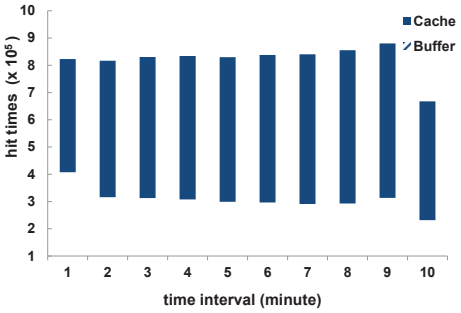

Fig. 9. Impact of prefetching buffer and HAC

5 Related Work

Recently, several key-value storage systems have sprung up in conjunction with flash storage such as BufferHash [22], SILT [24], FlashStore [23] and FAWN [12] etc. FAWN [12] is a power-efficient cluster architecture for data-intensive computing and uses SSDs as a replacement for HDDs. BufferHash [22] builds a content addressable memory system using flash for networking applications. It keeps the key-value pairs in the hash table and flushes the key-value pairs to flash when buffer is full. FlashStore [23] is a persistent key-value store using flash as a non-volatile cache between RAM and HDDs. SILT [24] is a memory-efficient key-value store by using partial-key cuckoo hashing and entropy-coded tries to reduce the per-key memory consumption. These key-value storage libraries are considered inserting SSD as another layer in the storage hierarchy to improve existing application performance. In contrast, MaiterStore treats the SSDs as an extension of the memory and pushes the SSDs upward in the memory hierarchy. Thus, using MaiterStore, existing shared-memory graph computation framework can easily and transparently enlarge the memory capacity to hundreds of gigabytes. Furthermore, by adopting the write-once-read-many model, we can easily eliminate many challenges associated with designing and implementing high-performance key-value store.

Apache Hadoop [1] is an open-source reincarnation of Google's MapReduce [26] that can be used for graph processing. It mainly consists of two components. The one is the MapReduce framework for processing the distributed data on large clusters inspired by the Bulk Synchronous Parallel model. The other is Hadoop Distributed File System (HDFS) [7], a distributed file system that is normally used by the MapReduce as its underlying storage for retrieving input and outputting results. Yet HDFS is inappropriate for storing the structured graph data in its current implementation, and MapReduce is inefficient for processing graph iterative computation. In contrast, MaiterStore can be used as the structured graph storage for efficiently supporting the Maiter graph processing. Moreover, MaiterStore has faster access speed than HDFS, because almost all the data are accessed in memory.

GraphChi [27] is a disk-based system on a single machine that can efficiently perform advanced computation on billion-node graphs. Even fast, it uses a sophisticated data structure that consists of partitions of the graph called 'shard'. Moreover, if there exists excessive iterations, GraphChi may involve a high number of I/O operations when updated edges are flushed to disk at each iteration. Whereas, MaiterStore aims to mitigate the memory pressure with a simple design by separating huge graph data from memory. Furthermore, we need not worry about high I/O latency, because the graph data on SSDs is non-volatile and read-only, and MaiterStore performs a minimal portion of data reads on SSDs.

Spark [18], Piccolo [25] and GraphLab [9, 16] are the graph iterative computation frameworks and they keep graph data in memory to achieve high computation speed. However, for huge graphs that do not fit in memory, such share-memory approach is constrained. Unlike these in-memory frameworks, MaiterStore separates the static graph-structured data from the memory, and mitigates the memory pressure. We need not worry about the Maiter performance of a sharp slowdown after using MaiterStore as the graph storage, because in MaiterStore the times of accessing SSDs count a surprisingly small percentage of all the data access.

6 Conclusion

This paper presents the design and evaluation of MaiterStore. MaiterStore is specialized for large-scale graph processing framework in the cloud. For efficiently managing the graph data, we separate the immutable key-value pairs from the in-memory state table in Maiter and store them on SSDs. In order to reduce the number of the SSD read operations and accelerate the graph algorithm execution, we adopt the page-based prefetching technique for buffering the to-be-queried data, and propose a hot-aware cache (HAC) for caching the hot and performance-critical data on SSDs. Our results show that MaiterStore is able to support graph processing more efficiently. Our ongoing work aims at extending MaiterStore with a more general I/O model and expects to apply MaiterStore to other in-memory graph processing frameworks beyond Maiter.

References

1. Hadoop,http://hadoop.apache.org.
2. Hama,http://hama.apache.org.
3. Amazon EC2, http://aws.amazon.com/ec2/.
4. Web Graph, http://lemurproject.org/clueweb09/.
5. Stanford dataset collection, http://snap.stanford.edu/data.
6. Samsung SSD, http://www.samsung.com/cn/business/business-products/ssd-card.
7. HDFS, http://hadoop.apache.org/core/docs/r0.16.4/hdfsdesign.html.
8. F.Chen, D. Koufaty and X. Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In ICS, 2011.

9. Low, Y., Gonzalez, A., Bickson, D., Guestrin, C., Hellerstein, J.M. Distributed Graphlab: A framework for machine learning in the cloud. In VLDB, 2012.
10. Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo and S. Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In ICS, 2011.
11. S. W. Lee, B. Moon, C. Park, J. M. Kim and S. W. Kim. A case for flash memory SSD in enterprise database applications. In SIGMOD, pages 1075–1086, 2008.
12. D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of WimpyNodes. In SOSP, Oct 2009.
13. S. Brin and L. Page, The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems, pages 107–117, 1998.
14. 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 OSDI, 2006.
15. M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In ACM Transactions on Computer Systems, 10(1), 1992.
16. Y.Low, J.Gonzalez, A.Kyrola, D.Bickson, C.Guestrin, and J.M.Hellerstein. Graphlab: A new framework for parallel machine learning. In UAI, 2010.
17. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In SIGMOD, pages 135–146, 2010.
18. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In HotCloud, 2010.
19. B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In ACM SIGMOD, 2013.
20. Y.Zhang, Q.Gao, L.Gao, and C.Wang. PrIter: A distributed framework for prioritized iterative computations. In SOCC, 2011.
21. Y.Zhang, Q.Gao, L.Gao, C.Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. In IEEE Computer Society, 2013.
22. A. Anand, C. Muthukrishnan, S. Kappes, A. Akella and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In NSDI, 2010.
23. B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. In VLDB, 2010.
24. H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In SOSP, 2011.
25. R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In OSDI, 2010.
26. J.Dean and S.Ghemawat. MapReduce: Simplified data processing on large clusters. In OSDI, 2004.
27. A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In OSDI, 2012.
28. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In SIGCOMM, 1999.
29. U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In KDD, pages 1091–1099, 2011.
30. S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In SIGMOD, 2009.
31. J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Math. 6, pages 29–123, 2009.