An I/O-Efficient Buffer Batch Replacement Policy for Update-Intensive Graph Databases

Ningnan Zhou[1,2], Xuan Zhou*[,1,2], Xiao Zhang[1,2], Shan Wang[1,2], and Ling Liu[3]

MOE Key Laboratory of DEKE, Renmin University of China
School of Information, Renmin University of China, Beijing, 100872
College of Computing, Georgia Institute of Technology
*Corresponding Author: zhou.xuan@outlook.com

Abstract. With the proliferation of graph based applications, such as social network management and Web structure mining, update-intensive graph databases have become an important component of today's data management platforms. Several techniques have been recently proposed to exploit locality on both data organization and computational model in graph databases. However, little investigation has been conducted on buffer management of graph databases. To the best of our knowledge, current buffer managers of graph databases suffer performance loss caused by unnecessary random I/O access. To solve this problem, we develop a novel batch replacement policy for buffer management. This policy enables us to maximally exploit sequential I/O to improve the performance of graph database. To enable the policy, we devise a segment tree based buffer manager to efficiently maintains optimal replacement plan. Extensive experiments on real-world and synthetic datasets demonstrate the superiority of our method.

Keywords: batch replacement, buffer manager, graph database, data manipulation, graph algorithm

1 Introduction

The rapid growth of graph data fosters a market of specialized graph databases such as Neo4j [9], Titan [10] and DEX [19]. To meet the needs of various graph based applications [11, 12, 14, 26–28, 34], these disk-based graph databases offer both database functionality such as insert/delete/update and analytical graph algorithms such as PageRank computation [6]. The evolving social network and the nature of some graph algorithms require graph databases to be update-friendly and update-efficient. For instance, to maintain a social network, each time a new friendship / connection establishes, a link connecting the pair of users should be inserted into the graph to reflect the change. In PageRank computation, the ranking score of every vertex needs to be updated in each iteration. This paper focuses on such update-intensive applications.

To support large scale graph databases, existing research work has mainly investigated the data organization and computational models. To achieve efficient

data organization, the associated edges of each vertex are normal stored together. For example, in social networks, the friends of a user are usually stored in continuous data pages [9]. As a result, frequent requests such as "return the friends of a specific user" in Facebook or Twitter [15] can benefit from low latency of sequential I/O. As to computational model, the dominant vertex-centric [18] or edge-centric [24] processing models partition a graph based on vertices or edges, and treat each partition as a unit of computation. They can also benefit from sequential I/O.

Although existing graph databases widely adopt I/O efficient data organization and computational models, they rarely consider buffer replacement policies. In fact, they still adopt variants of Least Recently Used (LRU) or Least Frequently Used (LFU) policies [7, 20], which evict one buffer page at a time and thus to some degree cancel out the effects of the specialized data organization and computational models. Figure 1 illustrates such a scenario. After the insertion of some new friends of user u, the data pages containing u's information, b_{u_1}, b_{u_2} and b_{u_3}, will be cached in the buffer. Note that b_{u_1}, b_{u_2} and b_{u_3} should be continuously located on disk. When a query such as "return the friend list of user v" is issued, the buffer manager requires to read in a new set of continuously located data pages, v_1, v_2 and v_3, which contain the friends of the user v. As the buffer is currently full, the buffer manager decides to evict b_{u_1}, b_{u_2} and b_{u_3} to make room for the incoming data pages. Following the existing replacement policy, the system will first seek to the position of u_1 to evict b_{u_1} and then seek to the position of v_1 to read in a new page. Iteratively, the systme will perform 6 random I/Os according to the order marked by the arrows in Figure 1. This is inefficient. If we can evict b_{u_1}, b_{u_2} and b_{u_3} in a batch, and read in v_1, v_2 and v_3 in a batch, we only need to perform two random disk seeks, and the other I/Os can be performed sequentially. Thus, such batch replacement can save 4 out of 6 random I/Os.

In this paper, we propose a batch replacement buffer manager for update-intensive graph databases. To the best of our knowledge, it is the first buffer replacement policy that exploits sequential I/O to speed up graph databases. Our design considers the following aspects: 1) the buffer manager should provide an unchanged interface to other layers of the graph database; 2) it should figure out the optimal replacement plan each time it needs to replace buffered pages; 3) it should minimize computational and memory overhead. To address these challenges, we first define the optimal replacement plan as the criteria to evict pages via sequential I/O. Then, we propose a segment tree based structure to organize buffered pages and to efficiently generate the optimal replacement plan. To evaluate the performance of our batch replacement buffer manager, we tried it on both real-world and synthetic datasets using typical workloads of database manipulation and graph algorithms. The experiment results show that 1) the batch replacement policy is able to achieve significant performance improvement by exploiting sequential I/O and 2) it is practical for graph databases.

The contributions of this paper are threefold:

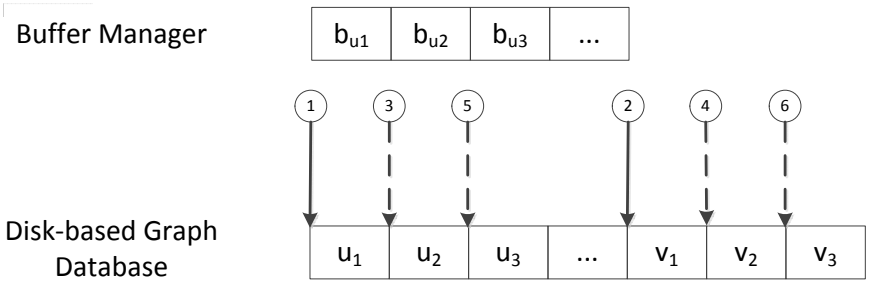


Fig. 1. An illustrative example for the effect of existing buffer manager and batch replacement in terms of random access, where the dashed arrow indicates the additional random access performed by existing buffer managers.

- We show the importance of exploiting sequential I/O in buffer management of graph databases.
- We propose a batch buffer replacement policy. Based on it, we define the optimal replacement plan and devise a segment tree based structure to manage buffered data pages and efficiently maintain the optimal plan.
- We conduct extensive experiments on real-world and synthetic datasets to verify the effectiveness of the batch replacement policy.

2 Related Work

Our work builds upon the existing techniques of graph databases, especially their data organization and computational models.

2.1 Data Organization

Conventionally, graph organization is built on top of the relational (*a.k.a.*, SQL) storage and graphs are stored as triplets [5, 25]. In other words, each edge e directed from a vertex u to a vertex v in the graph is transformed into a triplet $\langle u, e, v\rangle$. However, it is known that RDBMS organization is not good at answering traversal types of graph queries [30]. Considering the locality of data manipulation, such as queries like "return the friends of a specific user", it is more efficient to pack in-edges and out-edges of the same vertex in two lists and store them together [32, 22]. This has been adopted by most disk-based graph databases such as Neo4j. Therefore, we also assume such graph specific data organization.

2.2 Computational Model

Recently, a general iterative framework is adopted to process various graph algorithms such as PageRank and Shortest Path Computation. In the framework,

every vertex and edge in the graph is associated with a value and at each iteration, the value on a vertex or an edge is updated in vertex-centric or edge-centric model.

Vertex-centric model. Vertex-centric model is explored by initial works such as GraphLab [16] and Pregel [18]. In vertex-centric model, each vertex and its associated edges are regarded as a unit of computation so that if the main memory can hold any single vertex and its associated edges, only sequential I/O for loading data and updating results is required for each computation unit. To improve scalability, MOCgraph further reduces the memory footprint using message online computing [33].

Edge-centric model. Because a single vertex in real-world graph data, such as a celebrity, may be associated with so many edges that they cannot fit in main memory, edge-centric model is proposed [24, 14]. Edge-centric model partitions edges into disjoint sets and each set and its associated vertices form the unit of computation. In this way, each set can be hold in main memory to avoid random I/O access [34, 35, 13].

There is a significant body of work on distributed graph databases [23, 8, 29]. As our work focuses on speeding up a disk-based graph database on a single machine, our research is orthogonal and complementary to them.

2.3 Buffer Manager on Database

Existing buffer managers in graph databases usually adopt the variants of the LRU/LFU policy to reduce disk I/O. Neo4j adopts the LRU policy [9] while TurboGraph [13] maintains frequently used pages in memory. These works follow the same paradigm – when the buffer manager requires to read in a new page and the buffer gets overflow, only one buffered data page is evicted at a time. As a result, it introduces unnecessary random I/Os. To deal with this drawback, one recent work has proposed to remove buffer managers [17]. Besides, there are also alternative approaches which utilize index structures such as log structured merge tree [21] or fractal tree [3] to handle update-intensive workload. Both index structures process updates in a key range in a batch. However, as the physical pages of a key range may not be located consecutively on disk, random I/O still cannot be avoided completely.

In this paper, we aim to leverage sequential I/O by evicting buffered pages in a batch way rather following the existing paradigm which repeats evicting and reading one page at a time. Thus, our approach can benefit from the data organization and computational models for graph databases.

3 Batch Replacement Buffer Manager

In this section, we first present the problem definition for our batch replacement buffer manager. Then, we present the structure and algorithms of the proposed buffer manager.

3.1 Problem Formulation

As we have shown in Figure 1 in Section 1, it is inefficient to follow the existing paradigm of buffer manager, which evicts only one buffered data page at a time. In this paper, we extend the single page based replacement plan to the one that considers a set of pages. Thus, the new definition of replacement plan subsumes that of the existing buffer managers.

Definition 1. *Replacement Plan. When the buffer manager gets overflow, a replacement plan is a set of buffered data pages that will be evicted before the buffer manager performs any subsequent read operation.*

For example, the ideal replacement plan in Figure 1 is $\{b_{u_1}, b_{u_2}, b_{u_3}\}$.

Observing that evicting continuous buffered dirty data pages can maximize sequential I/O, the ideal batch replacement plan is to evict the longest sequence of such data pages.

Definition 2. *Optimal Batch Replacement Plan. Given a set of buffered pages with positions on the disk as $\mathcal{S} = \{p_1, p_2, ..., p_n\}$, the optimal batch replacement plan is a subset $\mathcal{P} \subseteq \mathcal{S}$ satisfying the following two conditions:*

1) pages in $\mathcal{P}$ are continuous in disk, namely, there are $n-1$ pairs of p_i and p_j in $\mathcal{P}$, such that $p_i \rightarrow p_j$ or $p_j \rightarrow p_i$, where $p_i \rightarrow p_j$ means that p_j is the successor data block in disk to p_i.

2) any other subset $\mathcal{P}' \subseteq \mathcal{S}$ satisfying Condition 1 contains less data pages than $\mathcal{P}$, namely, $|\mathcal{P}'| < |\mathcal{P}|$.

For example, in Figure 1, the optimal batch replacement plan is $\{p_{b_1}, p_{b_2}, p_{b_3}\}$. Although its subset such as $\{p_{b_1}, p_{b_2}\}$ satisfy the first condition, they violate the second condition and are not the optimal batch replacement plan.

3.2 Overview

We would like a buffer manager to change its replacement policy to the optimal batch replacement plan. However, we also prefer the change is transparent to other components of a graph database. We identify three properties the batch replacement buffer manager should possess: 1) *transparency* requires to export the same interface to other layers in a graph database; 2) *effectiveness* requires to identify the exact optimal replacement plan and 3) *efficiency* requires to minimize the computation and space cost of buffer manager.

When a data page is being updated, if it is surrounded by a number of continuous buffered dirty pages, batch replacement may evict such an active page and cause thrashing. Therefore, we use a "using" component to keep track of such active data pages to avoid them from being evicted. Although our batch replacement buffer manager is designed for update-intensive applications, we also need to ensure transparency for mixed workloads of read and write. Therefore, we use a "clear" component to keep track of unchanged data pages.

Besides the above-mentioned two components, the core component for our batch replacement buffer manager store all dirty data pages that can be evicted.

Figure 2 shows the transitions of a data page among the three components. Whenever the buffer manager reads a data page, it is inserted into the "using" component and only when the data page is unpinned and all queries referring to it terminate, it will be moved to the "clear" component or the core component, depending on if it has been updated. When the buffer overflows, the buffered data pages in the "clear" component will be evicted first. When the "clear" component is empty, the batch replacement plans will be used.

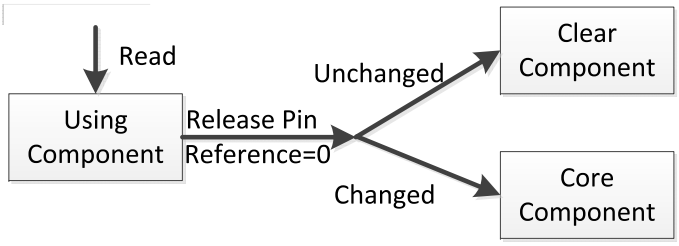


Fig. 2. The three components for Batch Replacement Buffer Manager.

To obtain an optimal replacement plan, the most straightforward approach is to sort all buffered data pages based on their positions in disk and then scan the sorted page list to find the longest continuous sequence. As shown in Algorithm 1, once we meet a continuous data page, we increase the length of the continuous page list (Line 7-10) and once the continuous data pages terminate, we update the replacement plan (Line 11-13). Although simple, this baseline algorithm is expensive, as it needs to sort and scan all buffered data pages.

3.3 Segment Tree based Buffer Manager

To avoid sorting and scanning, we adopt a segment tree based structure that maintains the buffered data pages that are continuous in disk [1]. In this way, each insertion routine actually amortizes the time for sorting and scanning.

To amortize the overhead of sorting, we represent each set of continuous data pages as an interval $[a, b]$, which indicates that these data pages start at the position a and end at the position b on disk. Note that such an interval represents individual data pages and continuous data pages in a unified way – the interval of an individual data page at position a on disk will be $[a, a]$. *To avoid the overhead of scanning*, we associate each interval with its interval length, on which the priority of eviction is based. In other words, the interval with the largest interval length will be chosen as the optimal replacement plan.

[1] For continence, the term "buffer manager" refers to the core component in the rest of the paper.

Algorithm 1 Trivial Algorithm

Require: $\mathcal{S} = \{p_1, p_2, ..., p_n\}$, the set of all buffered pages free to evict
Ensure: $\mathcal{P}$, the optimal replacement plan
1: Compute the list $\mathcal{L}$ by sorting pages in $\mathcal{S}$ in increasing order of positions in disk
2: $\mathcal{P} = \emptyset$
3: $len_{\mathcal{P}} = 0$
4: $\mathcal{P}' = \{\mathcal{L}[0]\}$
5: $len_{\mathcal{P}'} = 1$
6: **for** $i = 1$ to $n - 1$ **do**
7: **if** $\mathcal{L}[i-1] \rightarrow \mathcal{L}[i]$ **then**
8: $len_{\mathcal{P}'}++$
9: $\mathcal{P}' = \mathcal{P}' \cup \{\mathcal{L}[i]\}$
10: **else**
11: **if** $len_{\mathcal{P}'} > len_{\mathcal{P}}$ **then**
12: $\mathcal{P} = \mathcal{P}'$
13: $len_{\mathcal{P}} = len_{\mathcal{P}'}$
14: Return $\mathcal{P}'$

As Figure 3 illustrates, a segment tree is a balanced binary tree of height $O(\log n)$, using $O(n)$ space. It can support indexing of intervals with logarithmic computational complexity for insertion, deletion and querying [4]. Such a segment tree has the following 2 properties: 1) a key value is associated with each internal node. The intervals in its left branch end with positions no more than the key value and the intervals in its right branch start with positions larger than the key value; 2) an interval is associated with each internal node; it records the longest interval among all the intervals of its descendants.

For example, given the root node associated with the key value 14 and the interval $[5, 11]$, we know that: the interval $[17, 19]$ must be in its right branch because it starts at 17 which is larger than 14 (Property 1); the associated interval $[5, 11]$ is the longest interval in the buffer and its length is 7 (Property 2). In the figure, the interval $[14, 14]$ actually represents an individual data page at the position 14 on disk.

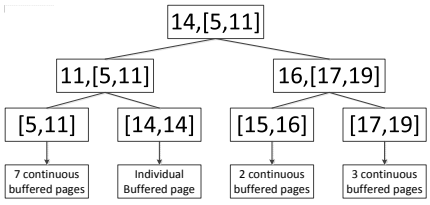


Fig. 3. An example segment tree, where leaf node represents intervals and internal node is associated with a key value and the longest interval among its descendants.

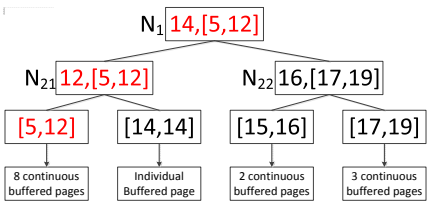


Fig. 4. The example segment tree after the page with position 12 at disk is inserted, where the updated nodes are marked in red.

Algorithm 2 Buffer Insert Algorithm

Require: d, the page to be inserted into the buffer
$tree$, the segment tree organizing buffered pages in the batch replacement buffer manager

```
1: New Interval new = [d.pos, d.pos]
2: Predecessor interval p = tree.search(d.pos − 1)
3: Successor interval s = tree.search(d.pos + 1)
4: if p exists then
5:     new = [p.start, d.pos]
6:     tree.delete(p)
7:     update longest intervals along the path from root to p
8: if s exists then
9:     new = [new.start, s.end]
10:    tree.delete(s)
11:    update longest intervals along the path from root to s
12: tree.insert(new)
13: update longest intervals along the path from root to new
```

The original segment tree is unable to maintain continuous data pages or the longest interval. It is our proposed insertion algorithm that utilizes the segment tree to maintain continuous data pages and the optimal replacement plan. The main idea is twofold: 1) whenever a buffered data page is inserted into the buffer manager, if its predecessor interval or successor interval exists, the inserted data page will extend the interval to a new longer interval and 2) whenever an interval is updated, the longest intervals on the path percolated from the root down to the interval itself will be updated. As Algorithm 2 illustrates, if the inserted data page d is at position $d.pos$ on disk, its predecessor interval should end with $d.pos - 1$ and its successor interval should start with $d.pos + 1$ (Line 2-3). If any one of the two intervals is found, it will be removed from the segment tree, and the intervals maintained by each internal node on the path from the root percolating to the interval will be updated (Line 7,11). Then, a new interval combining the predecessor/successor interval and the inserted data page will be inserted into the segment tree, and the longest intervals on the path from the root to the new interval will also be updated (Line 12-13). In this way, an insertion involves at most two queries, two deletions and one insertion on the segment tree. Thus its time complexity is $O(\log n)$, where n denotes the number of intervals and is normally less than the number of buffered data pages.

For example, given the segment tree in Figure 3, if we want to insert a page with position 12, we first find its predecessor interval $[5, 11]$, and combine it with the inserted page to form the new interval $[5, 12]$. Since no successor interval starting with $12 + 1 = 13$ is found in the segment tree, only the interval $[5, 11]$ is removed from the tree and the new interval is inserted. The longest intervals are updated correspondingly as marked in red in Figure 4.

Since the segment tree maintains the longest interval at the root node, whenever the buffer overflows, we simply pick up the data pages corresponding to the

longest interval as the optimal replacement plan. After the eviction, we can remove the corresponding interval and update the segment tree with amortized and worst case time complexity of $O(\log n)$. This procedure is efficient.

4 Experiment

In this section, we report experiment results on real-world and synthetic datasets. We demonstrate the effectiveness of our method on both database manipulation and graph algorithm execution. We also analyze the properties of the proposed batch replacement method.

4.1 Experimental Setting

Dataset. Two public real-world graph datasets were used, namely *Live Journal* [2] and *Friendster* [31]. Both datasets follow power-law distribution with parameter $\alpha \approx 1.4$, while the Friendster dataset is much larger than the Live Journal dataset. The parameter α controls the skewness of the power-law distribution, that is, with a small α such as 0.5, all vertices have similar number of edges, while with a large α such as 1.5, a small number of vertices have much more edges than others. The synthetic dataset is generated by *LinkBench*, the graph database benchmark published by Facebook [1]. It is able to generate graphs with power-law distribution under varying α. The detailed statistics are shown in Table 1.

Table 1. Statistics of our datasets.

Dataset	# Vertex	# Edges	Raw Size
Live Journal	$4,847,571$	$68,993,773$	$2.3GB$
Friendster	$65,608,366$	$1,806,067,135$	$150GB$
LinkBench	$10^6 \sim 10^7$	$10^8 \sim 10^9$	$5 \sim 60$GB

Workload. The workloads included typical graph algorithms and database manipulation. Following [14, 23, 17, 35], we ran typical graph algorithms including PageRank (PR), Single-Source Shortest Paths (SSSP), Weakly Connected Components (WCC) and Sparse Matrix Multiplication (SMM). LinkBench also provides a mix of insert/delete/update operations on vertices and edges as basic graph database manipulation.

All experiments were conducted on a machine with 2.5 Ghz Intel Core 2 CPU, 8GB of RAM and 10TB, 15,000 rpm hard drive. We implemented the proposed batch replacement buffer manager on Neo4j [2] (Neo4j-BR) and GraphChiDB [3] (ChiDB-BR). Neo4j is a leading industry-standard graph database that

[2] http://neo4j.com/
[3] https://github.com/graphchi/graphchiDB-scala

adopts LRU-based buffer manager and vertex-centric programming model, while GraphChi-DB (ChiDB) is a research prototype that discards buffer manager and adopts edge-centric programming model. For database manipulation, we also report the performance of a relational database MySQL, only for the purpose of reference. ChiDB also has an option to adopt log-structured merge tree (ChiDB-LSM) for write-optimized database manipulation. We explicitly created appropriate indexes for all databases during the experimental study.

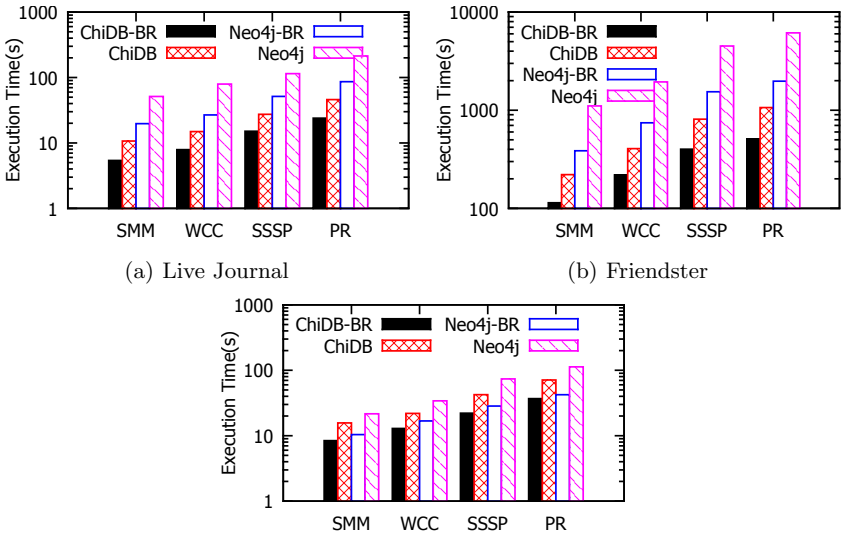
4.2 Performance Comparison

In this section, we first show the effectiveness of our batch replacement buffer manager for data manipulation and graph algorithms. Then, we show that our approach is robust for various buffer sizes and workloads.

Figure 5 shows the average execution time for the typical graph algorithms. The buffer size BS is set to 5% of the dataset size. We have three observations: 1) for all graph algorithms on all datasets, the batch replacement variants of the two graph databases outperform their original versions. This shows that our batch replacement policy is superior to the LRU-based policy and the approach that does not use buffer manager; 2) on both real-world datasets, ChiDB-BR and ChiDB outperforms Neo4j-BR and Neo4j. This shows edge-centric programming model is more suitable for graph algorithms on real-world datasets. The high value of $\alpha \approx 1.4$ indicates that a few vertices may contain a huge number of edges so that data pages involved in these vertices are read and evicted repeatedly in Neo4j and Neo4j-BR. Even though, our batch replacement policy exhibits better performance than the LRU-based policy; 3) on the synthetic dataset, Neo4j-BR outperforms ChiDB. This is because under $\alpha = 0.5$ edges are distributed more uniformly on vertices and thus Neo4j-BR benefit from less buffered page eviction.

Table 2 shows the average execution time for various manipulation workload on a small dataset (5GB) and a large dataset (50GB) respectively. We have the following observations: 1) on both datasets, both Neo4j-BR and ChiDB-BR outperform the original databases equipped with LRU-based buffer manager or log structure merge tree or no buffer manager; This indicates that batch replacement buffer manager is more suitable for graph databases; 2) Neo4j-BR and ChiDB-BR outperform MySQL, which shows the superiority of specialized graph database; 3) Neo4j outperforms ChiDB on small dataset while ChiDB outperforms Neo4j on large dataset, revealing that LRU-based buffer management is sensitive to the scale of dataset, while batch replacement buffer management is more robust.

Both batch replacement buffer manager and log structured merge tree are designed for update-intensive applications by leveraging sequential I/O. However, ChiDB-BR outperforms ChiDB-LSM in most cases. This is because LSM-tree does not consider the optimal replacement plan. Sometimes LSM-tree's data accesses will be scattered across a wide range on disk, which incurs numerous random I/Os.



(a) Live Journal

(b) Friendster

(c) LinkBench

Fig. 5. Execution Time for Graph Algorithms on Three Datasets, where the synthetic dataset contains 10^6 vertices and 10^8 edges with $\alpha = 0.5$. $BS = 5\%$ of dataset size.

Figure 6 validates the robustness of our approach on various ratios of buffer size to data size. On Live Journal dataset, we continuously increased the buffer size until the whole dataset was hold in main memory. The execution time of the PageRank algorithm keeps dropping. We can see: 1) until the buffer holds half the dataset, graph databases employing the batch replacement policy always outperform their counterparts; therefore, our approach can exploit available main memory efficiently; 2) when the buffer holds the whole dataset and buffer replacement is no longer needed, our approach consumes 1% less execution time than their counterparts; this shows that our method for identifying optimal replacement plans is efficient.

Figure 7 shows the query performance on the Friendster dataset for typical read-only workloads, including retrieval of a specific vertex / edge and a traversal-heavy Friends-of-Friends (FoF) query. The FoF query is defined to find all vertices which can reach a specific vertex via any proxy vertex. We can see that although maintaining intervals of continuous buffered pages is of no use since there is no replacement for dirty pages, the overhead is still low. Therefore, although our batch replacement buffer manager is designed for update-intensive applications, its performance is acceptable for read-only applications as well.

Table 2. Execution Time (ms) for Graph Database Manipulation on Synthetic Dataset with $\alpha = 1.5$ and $BS = 5GB$.

Data Size	Operation	ChiDB-BR	ChiDB	ChiDB-LSM	Neo4j-BR	Neo4j	MySQL
10^6 vertices, 10^8 edges	node_insert	0.09	12.9	0.10	**0.08**	0.13	0.11
	node_delete	0.10	16.7	0.14	**0.07**	0.12	0.17
	node_update	0.12	19.1	0.16	**0.09**	0.13	0.21
	edge_insert	0.15	24.6	0.17	**0.09**	0.19	0.25
	edge_delete	0.15	26.3	0.19	**0.12**	0.19	0.34
	edge_update	0.19	29.5	0.22	**0.14**	0.22	0.41
10^7 vertices, 10^9 edges	node_insert	**31**	94	37	36	259	42
	node_delete	**33**	105	41	39	268	45
	node_update	**34**	116	46	41	280	49
	edge_insert	**42**	136	55	47	295	64
	edge_delete	**48**	152	63	57	323	69
	edge_update	**51**	159	67	62	344	73

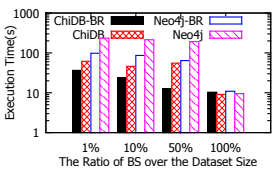


Fig. 6. Effect of RAM size on Live Journal.

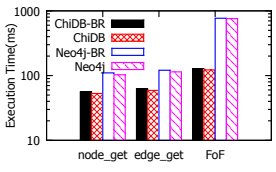


Fig. 7. Query Time on Friendst., BS=2GB.

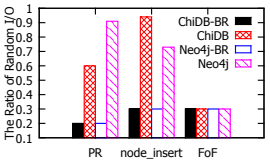


Fig. 8. Ratio of Random I/O access on Friendster dataset.

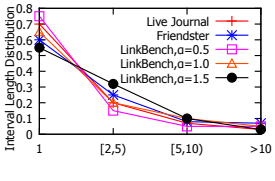


Fig. 9. Interval Length Distribution for PageRank.

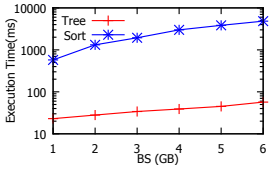


Fig. 10. CPU Time for Replacement Plan.

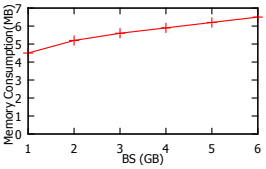


Fig. 11. Memory Overhead.

4.3 Property of Batch Replacement

In this section, we evaluate the effectiveness of our batch replacement policy in terms of I/O and the computational overhead.

Figure 8 plots the ratios of random I/O to all disk I/O for the workloads of PageRank, node insertion and FoF query respectively, which represent typical workloads of graph algorithm, database manipulation and read-only query. We can observe that both Neo4j-BR and ChiDB-BR used the least random I/O access. Therefore, it is not surprising their execution time is the shortest in aforementioned experiments. Figure 9 depicts the distribution of buffered interval lengths when running the PageRank Algorithm on the Friendster dataset. We can see that on most datasets there are sufficient segments of continuous buffered data pages. Therefore, it is always possible for our batch replacement buffer manager to exploit sequential I/Os. The distribution of random I/O and interval lengths for other graph algorithms and data manipulation are similar to Figure 8 and 9.

Figure 10 shows the average execution time for each batch replacement using our segment tree based solution (Tree) and the trivial sort-based algorithm (Sort, Algorithm 1) on the Friendster dataset for the PageRank Algorithm. We can see that as the buffer size increases, our segment tree based solution outperforms the trivial sort-based solution significantly. Figure 11 shows the additional memory consumption for maintaining the segment tree of continuous pages on the Friendster dataset for the PageRank Algorithm. We can see that the segment tree only consumes less than 1% of the buffer size. Note that the computational and memory overhead are normally only influenced by buffer size, rather than the variation of workloads and datasets.

5 Conclusion

In this paper, we propose a novel approach to batch replacement buffer management for graph databases. Taking the specific data organization and vertex-centric or edge-centric programming models into consideration, the proposed method enables graph databases to make the best of sequential I/O. In addition to a sort-based trivial solution to find optimal replacement plan, we propose a segment tree based buffer structure to efficiently maintain optimal replacement plans. Extensive experiments on real-world and synthetic datasets show that our approach significantly improve the performance of existing graph databases and outperforms the LRU-based approaches and a recently proposed no-buffer approach. The experiment results also show that our approach incurs minimum computational and memory overhead and therefore is practical for real-world applications.

Acknowledge

This work is partially funded by China Scholarship Council. Ling Lius research is partially supported by the National Science Foundation under Grants IIS-

0905493, CNS-1115375, IIP-1230740 and a grant from Intel ISTC on Cloud Computing.

References

1. Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. SIGMOD '13, pages 1185–1196.
2. Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. KDD '06, pages 44–54.
3. Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.
4. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008.
5. Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. SIGMOD '13, pages 121–132.
6. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
7. Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, December 1984.
8. Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. OSDI'14, pages 599–613.
9. Neo4j graph database. http://neo4j.com/.
10. Titan graph database. http://thinkaurelius.github.io/titan/.
11. Jialong Han and Ji-Rong Wen. Mining frequent neighborhood patterns in a large labeled graph. CIKM '13, pages 259–268.
12. Jialong Han, Ji-Rong Wen, and Jian Pei. Within-network classification using radius-constrained neighborhood patterns. CIKM '14, pages 1539–1548.
13. Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. KDD '13, pages 77–85.
14. Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46.
15. Twitter Developer:Get Friends List. https://dev.twitter.com/rest/reference/get/friends/list.
16. Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. PVLDB '12.
17. P. Macko, V.J. Marathe, D.W. Margo, and M.I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. ICDE '15, pages 363–374.
18. Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. SIGMOD '10, pages 135–146.

19. Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez, and Josep-L. Larriba-Pey. Dex: High-performance exploration on large graphs for information retrieval. CIKM '07, pages 573–582.
20. Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *J. ACM*, 46(1):92–112, January 1999.
21. Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.
22. Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.
23. Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. SOSP '15, pages 472–488.
24. Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, pages 472–488.
25. Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. BTW '13, pages 403–420.
26. Shuo Shang, Ruogu Ding, Bo Yuan, Kexin Xie, Kai Zheng, and Panos Kalnis. User oriented trajectory search for trip recommendation. EDBT '12, pages 156–167.
27. Shuo Shang, Ruogu Ding, Kai Zheng, Christian S. Jensen, Panos Kalnis, and Xiaofang Zhou. Personalized trajectory matching in spatial networks. *The VLDB Journal*, 23(3):449–468, 2014.
28. Shuo Shang, Bo Yuan, Ke Deng, Kexin Xie, Kai Zheng, and Xiaofang Zhou. Pnn query processing on compressed trajectories. *Geoinformatica*, 16(3):467–496, July 2012.
29. Bin Shao, Haixun Wang, and Yanghua Xiao. Managing and mining large graphs: Systems and implementations. SIGMOD '12, pages 589–592.
30. Yinglong Xia, I.G. Tanase, Lifeng Nai, Wei Tan, Yanbin Liu, J. Crawford, and Ching-Yung Lin. Graph analytics and storage. IEEE Big Data '14, pages 942–951.
31. Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. MDS '12, pages 3:1–3:8.
32. Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. PVLDB '13, pages 265–276.
33. Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. Mocgraph: Scalable distributed graph processing using message online computing. pages 377–388.
34. Yang Zhou, Ling Liu, Kisung Lee, and Qi Zhang. Graphtwist: Fast iterative graph computation with two-tier optimizations. PVLDB '15, pages 1262–1273.
35. Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. USENIX ATC '15, pages 375–386.