Data Sci. Eng. (2016) 1:231–241
DOI 10.1007/s41019-016-0026-9

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]

Received: 14 July 2016 / Accepted: 16 December 2016 / Published online: 11 January 2017
© The Author(s) 2017. This article is published with open access at Springerlink.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. However, trivial solution produces impractical maintenance for replacement plan with maximal sequential I/O. To enable the policy, we first devise a segment tree-based buffer manager to efficiently maintain a optimal replacement plan. Unfortunately, segment tree-based solution becomes bottleneck in multi-core environment. To remedy this weakness, a B-tree-based buffer manager is further proposed. Extensive experiments on real-world and synthetic datasets demonstrate the superiority of our method.

Keywords Batch replacement · Buffer manager · Graph database

✉ Xuan Zhou
zhou.xuan@outlook.com

1 MOE Key Laboratory of DEKE, Renmin University of China, Haidian, China

2 School of Information, Renmin University of China, Beijing 100872, China

1 Introduction

The rapid growth of graph data fosters a market of specialized graph databases such as Neo4j,[1] Titan[2] and DEX [16]. To meet the needs of various graph-based applications [12, 28], these disk-based graph databases offer both database functionality such as insert/delete/update and analytical graph algorithms such as PageRank computation [7]. 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 in neo4j. As a result, frequent requests such as "return the friends of a specific user" in Facebook or Twitter[3] can benefit from low latency of sequential I/O. As to computational model, the dominant vertex-centric [15] or edge-centric [21] 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.

1 http://neo4j.com/.
2 http://thinkaurelius.github.io/titan/.
3 https://dev.twitter.com/rest/reference/get/friends/list.

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 [8, 17], 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 system will perform 6 random I/Os according to the order marked by the arrows in Fig. 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.

Since there is no specific optimized concurrency control strategy for segment tree, our segment tree-based buffer manager suffers from concurrent updates. To remedy this weakness, we propose to transform the replacement plans into B-tree organization and thus it benefits from sophisticated B-tree concurrency control techniques.

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 fourfold:

- 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 propose to transform the replacement plans in segment tree-based buffer manager to B-tree organization. Consequently, our batch replacement buffer manager can benefit from sophisticated concurrency control techniques for B-tree.
- 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 [6, 22]. 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 [24]. 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 [19, 26]. 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 are 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.

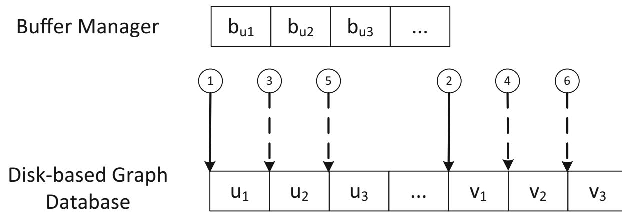

Fig. 1 An illustrative example for the effect of existing buffer manager and batch replacement in terms of random access, where the dashed arrow indicates the additional random access performed by existing buffer managers

2.2.1 Vertex-centric Model

Vertex-centric model is explored by initial works such as GraphLab [13] and Pregel [15]. 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 [27].

2.2.2 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 [12, 21]. 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 [10, 28, 29].

There is a significant body of work on distributed graph databases [9, 20, 23]. 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 while TurboGraph [10] 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 [14]. Besides, there are also alternative approaches which utilize index structures such as log structured merge tree [18] or fractal tree [4] 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 Fig. 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 Fig. 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 Fig. 1, the optimal batch replacement plan is $\{p_{b_1}, p_{b_2}, p_{b_3}\}$. Although its subset such as $\{p_{b_1}, p_{b_2}\}$ satisfies the first condition, they violate the second condition and are not the optimal batch replacement plan.

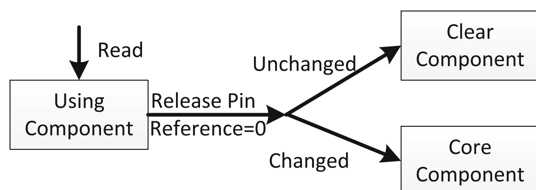

Fig. 2 The three components for batch replacement buffer manager

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.

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.

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 $\mathcal{P}'$

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.[4] 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

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

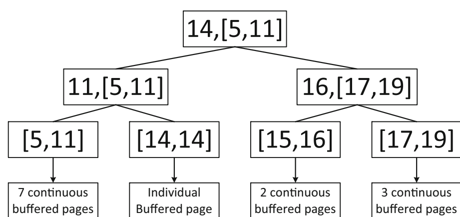

Fig. 3 An example segment tree, where leaf node represents intervals and internal node is associated with a key value and the longest interval among its descendants

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.

As Fig. 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 [5]. 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.

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 Fig. 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

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

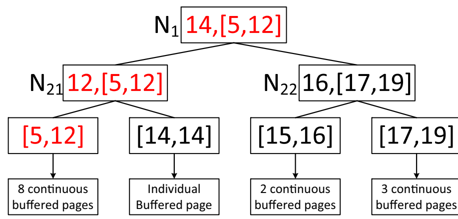

Fig. 4 The example segment tree after the page with position 12 at disk is inserted, where the updated nodes are marked in *red*

tree and the new interval is inserted. The longest intervals are updated correspondingly as marked in red in Fig. 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 Concurrency Control

Our segment tree-based buffer manager suffers from concurrent updates from two aspects: (1) lack of optimized concurrency control strategies and (2) the binary structure of segment tree reduces granularity of concurrency control. To this end, we propose to transform all the replacement plans hold by the segment tree into B-tree organization. In this way, we can use the concurrency sophisticated B-tree to handle concurrent updates on the buffer manager. In this paper, we adopt a sophisticated multi-version B-tree implementation [3]. Note that any concurrency-supporting B-tree can be adopted such as the recently proposed multi-core environment specific Bw tree [11]. In the following, we first describe the transformation from the segment tree-based buffer manager to B-tree organization and then present how to maintain optimal replacement plan.

The transformation from the segment tree structure to a B-tree organization is based on the following observation: each candidate replacement plan, e.g., continuous pages in disk, is regarded as an interval and all candidate plans are disjoint. For example, there are totally 4 candidate replacement plans in Fig. 3, namely [5,11], [14,14], [15,16] and [17,19]. These intervals can also be represented in key-length pairs in the form of $\langle k, l\rangle$, where the key k denotes the start position of each candidate plan in disk and the length l indicates the number of pages that can be flushed in sequential I/O. For example, these 4 candidate replacement plans can be represented by four key-length pairs $\langle 5,7\rangle$, $\langle 14,1\rangle$, $\langle 15,2\rangle$ and $\langle 17,19\rangle$. In this way, each replacement plan can be organized in B-tree in a natural way. As Fig. 5 illustrates, the B-tree indexes the keys in each key-length pairs and the length is stored in leaf nodes. Similar to the segment tree-based buffer manager, each internal node of the B-tree also refers to the optimal replacement plan underlying itself. For example, the root node contains three keys separating the four candidate replacement plans and indicates that the candidate plan with start position less than 14 is the optimal replacement plan.

When a new page is buffered, it will be merged into one existing replacement plan or become an individual replacement plan. Since the new page with position p in disk can be merged into an existing replacement plan if and only if it is the successor or predecessor of an existing key-length pair, we search two potential key-length pairs $\langle k_1, l_1\rangle$ and $\langle k_2, l_2\rangle$ in the B-tree such that (1) $k_1 < p$ and $\forall k' < p, k_1 \geq k'$ and (2) $k_2 > p$ and $\forall k' > p, k_2 \leq k'$. The first condition is the necessary condition that the new page is a successor of the replacement plan $\langle k_1, l_1\rangle$, and the second condition is the necessary condition that the new page is a predecessor of the replacement plan $\langle k_2, l_2\rangle$. If the new page is a successor of the replacement plan $\langle k_1, l_1\rangle$, it should hold that $p = k_1 + l$; if the new page is a predecessor of the replacement plan $\langle k_2, l_2\rangle$, it should hold that $p + 1 = k_2$.

The maintenance under B-tree is similar to segment tree-based maintenance, and in the following, we use an example to reveal the details. As Fig. 6 illustrates, given the original B-tree shown in Fig. 5, when a new page with position 12 becomes free to evict, B-tree first finds that $\langle 5,7\rangle$ is the first replacement plan with start position less than the new page position in disk and $\langle 14,1\rangle$ is the first replacement plan with start position greater than the new page position in disk. For the key-length pair $\langle 5,7\rangle$, we can determine the new page is the successor of this replacement plan because $p = k_1 + l$, where $p = 12$, $k_1 = 5$ and $l = 7$. Therefore, the new page is merged into a new replacement

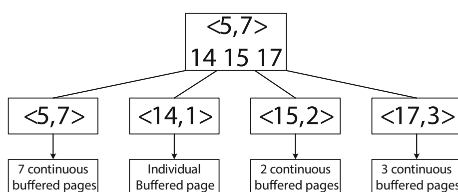

Fig. 5 An example for replacement plans organized by B-tree

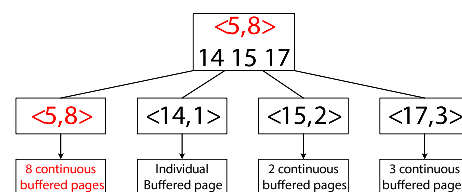

Fig. 6 An example for replacement plan maintenance in B-tree

Table 1 Statistics of our datasets

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

plan. Meantime, in each internal node up toward the root node, the optimal replacement plan is updated.

5 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.

5.1 Experimental Setting

5.1.1 Dataset

Two public real-world graph datasets were used, namely *Live Journal* [2] and *Friendster* [25]. 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* and 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.

5.1.2 Workload

The workloads included typical graph algorithms and database manipulation. Following [12, 14, 20, 29], 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[5] (Neo4j-BR) and GraphChi-DB[6] (ChiDB-BR). Neo4j is a leading industry standard graph database that 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.

5.2 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 7 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 outperform 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. However, 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-

[5] http://neo4j.com/.

[6] https://github.com/graphchi/graphchiDB-scala.

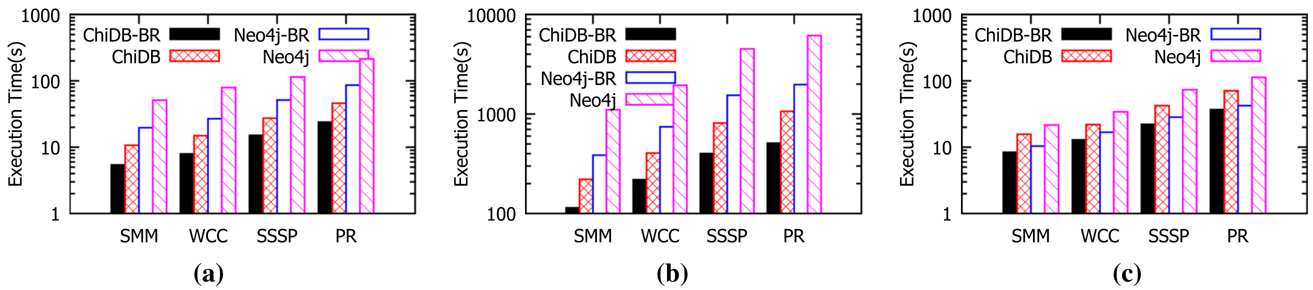

Fig. 7 Execution time for graph algorithms on three datasets, where the synthetic dataset contains 10^6 vertices and 10^8 edges with $\alpha = 0.5$. *BS*= 5% of dataset size. **a** Live Journal, **b** Friendster, **c** LinkBench

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

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.

Figure 8 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.

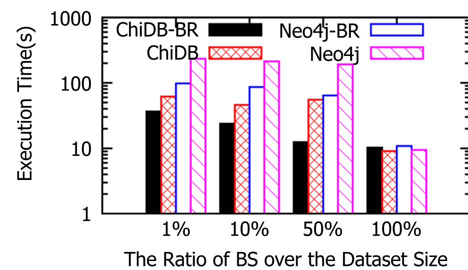

Fig. 8 Effect of RAM size on Live Journal

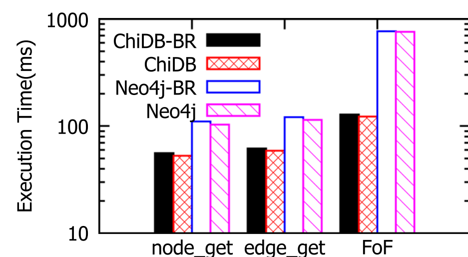

Fig. 9 Query time on Friendst., *BS* = 2GB

Figure 9 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-

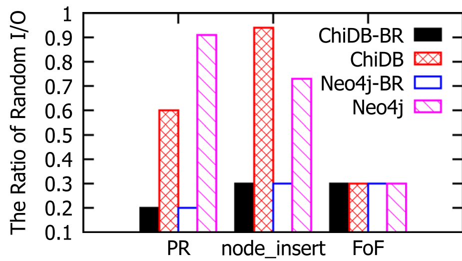

Fig. 10 Ratio of random I/O access on Friendster dataset

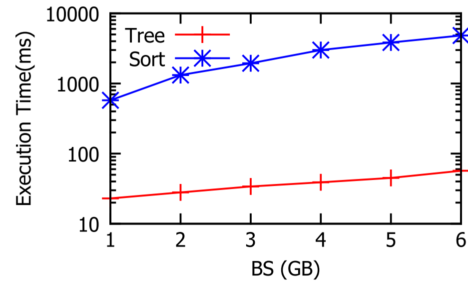

Fig. 12 CPU time for replacement plan

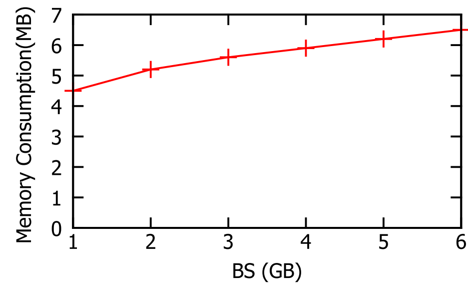

Fig. 13 Memory overhead

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.

Figure 14 compares the performance under different number of threads. We can see that on graph algorithms, the performance of all methods does not increases much because processing is blocked by I/O. Instead, for data manipulations, we can see that our B-tree-based buffer manager does not incur performance drop and still outperforms other methods under parallel processing, while the segment tree-based method suffers from high concurrency no matter which concurrency control is adopted.

5.3 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 10 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 11 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 Figs. 10 and 11.

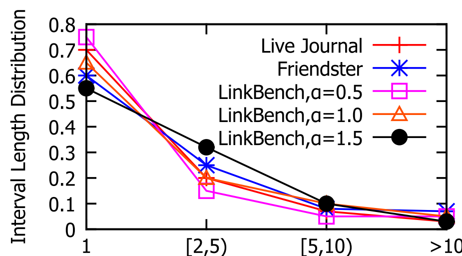

Fig. 11 Interval length distribution for PageRank

Figure 12 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 13 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 (Fig. 14).

6 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

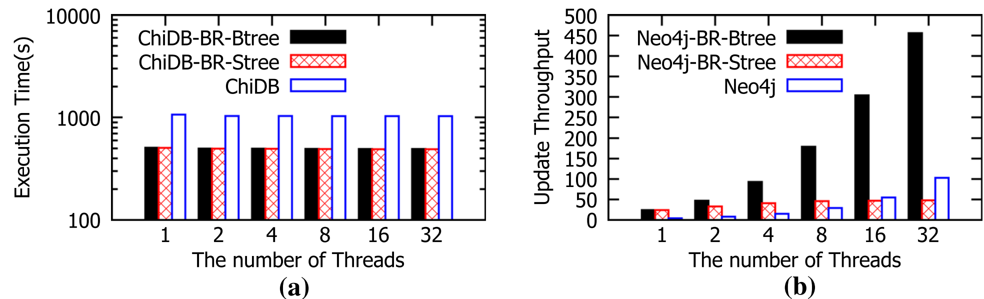

Fig. 14 Effect of multi-threads on Friendster dataset with BS = 5% of dataset size **a** Page Rank, **b** data manipulation

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. To utilize multi-core environment, we propose to transform all replacement plans in segment tree-based buffer manager into B-tree organization. As a result, our batch replacement buffer manager benefits from sophisticated concurrency control techniques for B-tree. Extensive experiments on real-world and synthetic datasets show that our approach significantly improves 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.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M (2013) Linkbench: a database benchmark based on the facebook social graph. SIGMOD, pp 1185–1196
2. Backstrom L, Huttenlocher D, Kleinberg J, Lan X (2006) Group formation in large social networks: membership, growth, and evolution. KDD, pp 44–54
3. Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P (1996) An asymptotically optimal multiversion b-tree. VLDB J 5(4):264–275
4. Bender MA, Demaine ED, Farach-Colton M (2005) Cache-oblivious b-trees. SIAM J Comput 35(2):341–358
5. Berg M, Otfried C, Marc van K, Mark O (2008) Computational geometry: algorithms and applications, 3rd edn. Springer, Berlin
6. Bornea MA, Dolby J, Kementsietsidis A, Srinivas K, Dantressangle P, Udrea O, Bhattacharjee B (2013) Building an efficient rdf store over a relational database. SIGMOD, pp 121–132
7. Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. Comput Netw 30(1–7):107–117
8. Effelsberg W, Haerder T (1984) Principles of database buffer management. ACM Trans Database Syst 9(4):560–595
9. Gonzalez JE , Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) Graphx: graph processing in a distributed dataflow framework. OSDI, pp 599–613
10. Han WS, Lee S, Park K, Lee J-H, Kim M-S, Kim J, Yu H (2013) Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. KDD, pp 77–85
11. Sengupta S, Levandoski J, Lomet D (2013). The bw-tree: a b-tree for new hardware. ICDE
12. Kyrola A, Blelloch G, Guestrin C (2012) Graphchi: large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX conference on operating systems design and implementation, OSDI, pp 31–46
13. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A (2012) Hellerstein JM Distributed graphlab: a framework for machine learning and data mining in the cloud. PVLDB 5(8):716–727. doi:10.14778/2212351.2212354
14. Macko P, Marathe VJ, Margo DW, Seltzer MI (2015) Llama: Efficient graph analytics using large multiversioned arrays. ICDE, pp 363–374
15. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G(2010) Pregel: a system for large-scale graph processing. SIGMOD, pp 135–146
16. Martínez-Bazan N, Muntés-Mulero V, Gómez-Villamor S, Nin J, Sánchez-Martínez MA, Larriba-Pey JL (2007) Dex: high-performance exploration on large graphs for information retrieval. CIKM, pp 573–582
17. O'Neil EJ, O'Neil PE, Weikum G (1999) An optimality proof of the lru-k page replacement algorithm. J ACM 46(1):92–112
18. O'Neil P, Cheng E, Gawlick D, O'Neil E (1996) The log-structured merge-tree (lsm-tree). Acta Inf 33(4):351–385
19. Robinson I, Webber J, Emil E (2013) Graph databases. O'Reilly Media, Inc, Sebastopol
20. Roy A, Bindschaedler V, Malicevic J, Zwaenepoel W (2015) Chaos: scale-out graph processing from secondary storage. SOSP, pp 472–488
21. Roy A, Mihailovic I, Zwaenepoel W (2013) X-stream: edge-centric graph processing using streaming partitions. SOSP, pp 472–488
22. Rudolf M, Paradies M, Bornhövd C, Lehner W (2013) The graph story of the SAP HANA database. BTW, pp 403–420
23. Shao B, Wang H, Xiao Y (2012) Managing and mining large graphs: systems and implementations. SIGMOD, pp 589–592
24. Xia Y, Tanase IG, Nai L, Tan W, Liu Y, Crawford J, Lin CY (2014) Graph analytics and storage. IEEE Big Data, pp 942–951
25. Yang J, Leskovec J (2012) Defining and evaluating network communities based on ground-truth. MDS, pp 3:1–3:8
26. Zeng K, Yang J, Wang H, Shao B, Wang Z (2013) A distributed graph engine for web scale rdf data. PVLDB, pp 265–276
27. Zhou C, Gao J, Sun B, Yu JX (2014) Mocgraph: scalable distributed graph processing using message online computing. pp 377–388

28. Zhou Y, Liu L, Lee K, Zhang Q (2015) Graphtwist: fast iterative graph computation with two-tier optimizations. PVLDB, pp 1262–1273
29. Zhu X, Han W, Chen W (2015) Gridgraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. ATC, pp 375–386