Locality-Aware Cache Replacement Policy for Graph Traversals

Zeynep Korkmaz
University of Waterloo
Waterloo, ON, Canada
zkorkmaz@uwaterloo.ca

M. Tamer Özsu
University of Waterloo
Waterloo, ON, Canada
tamer.ozsu@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
Waterloo, ON, Canada
khuzaima.daudjee@uwaterloo.ca

ABSTRACT

Many graph processing applications consist of read-only workloads that need to perform low-latency traversals over large graphs. These traversals are inherently expensive, and storage and processing systems need to be optimized for them. The performance of secondary storage-based systems can be improved by caching locality-driven data in memory. Exploring the data reuse of graph objects in applications is important to decrease the page faults in the cache. However, graph applications can suffer from poor access locality, making caching of graph data challenging. Locality can be imposed through graph ordering algorithms that can be exploited by cache replacement algorithms. We propose a graph locality-aware cache replacement policy called LAC that exploits the serialization layout obtained by graph ordering techniques. We show that the spatial locality that is captured on disk pages offers temporal locality for subsequent accesses of cache pages, and this information can be used to make improved cache replacement decisions. We evaluate LAC against the popular GCLOCK algorithm for input graphs with different structural properties while running various query types. Our evaluation shows that LAC can outperform GCLOCK through page fault improvements by reducing latency up to 1.42× in simulation studies and up to 1.23× with integration into the Neo4j system.

PVLDB Reference Format:
Zeynep Korkmaz, M. Tamer Özsu, and Khuzaima Daudjee. Locality-Aware Cache Replacement Policy for Graph Traversals. PVLDB, 18(9): 2859 - 2871, 2025.
doi:10.14778/3746405.3746413

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zeynepsaka/graphs-LAC.

1 INTRODUCTION

Graphs are popularly used to represent and model many real-world relationships among data items [25]. Vertices in these graphs represent entities (e.g., people, processes, assets, devices), and the edges represent the relationships among these entities. The volume of graph-structured data as well as the sizes of these graphs continue to grow, tracking the enormous growth in general data volume [14]. Many graph processing applications consist of read-only workloads that require traversals over large graphs. To support low-latency traversals, there is a need for graph storage and processing systems that can support these traversals efficiently [33]. In this vein, graph DBMSs (GDBMS) have been an active area of research and development, although their performance and scalability continue to be a major source of concern for users [25, 26]. The growth of graph data sizes can outstrip memory capacities, requiring this data to be stored on secondary storage.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746413

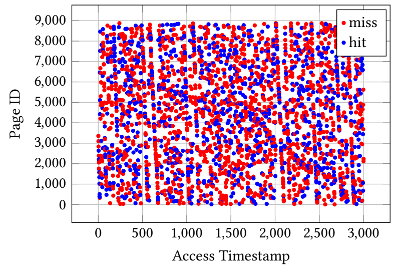


Figure 1: Temporal data accesses of disk pages in the cache when the graph is serialized without structure awareness.

Data caching plays an important role in the performance of these disk-based[1] systems. Caches facilitate retaining frequently and/or recently accessed data in memory, lowering data access latencies by enhancing data locality. Satisfying application data requests through the cache reduces the number of page faults required to fetch data from the disk. Since the operating system is agnostic to graph data access patterns, cache management policies that can leverage graph data access patterns can improve cache performance, and therefore system performance. However, the absence of access locality in graph applications is a challenge [18, 32].

Figure 1 is a plot of temporal data accesses (hits and misses) of disk pages in the cache that contain the vertices of the input graph serialized on disk by using the storage layout of one of the popular GDBMSs [1][2] when running single pair shortest path queries on social network graph data. The scattered access pattern of pages in the cache shows a divergence of the physical data order on disk from the traversal access pattern. This is due to serialization on disk without considering the graph's topology. In this experiment the well-known and widely-used cache replacement policy, GCLOCK [27] is deployed; GCLOCK considers recency and frequency of page accesses in making its decisions. This scattered access problem can be addressed by improving data locality through algorithms that serialize data on disk such that accesses to these data correlate with query traversals.

[1]In this paper, we simply use the term "disk-based" to refer to any persistent secondary storage.
[2]The storage engine stores edges and vertices of the input graph separately both on disk and in cache pages.

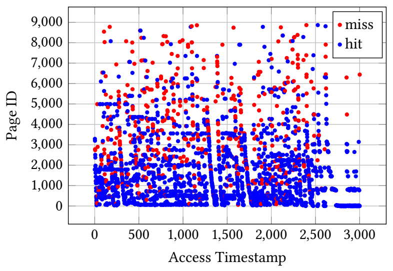


Figure 2: Temporal data accesses of disk pages in the cache when the graph is serialized with structure awareness.

Figure 2 shows a plot of the same graph data and query as in Figure 1 but with a changed data layout that correlates with the graph topology, e.g., adjacent vertices and neighbourhoods in the graph are stored on the same page or on adjacent pages on disk by using a topology-aware serialization approach [29]. The resulting improved locality of data accesses generates more cache hits as well as more clustered and consecutive misses.

The clustered and consecutive misses in the plot show that data reuse in graph applications is complex and generic cache replacement techniques that are currently deployed in GDBMSs do not fit the access patterns in graphs. This motivates the need for caching techniques that can support this topology-based locality to improve query performance. Although it has been shown that serialization that takes into account graph topology can improve intra-page spatial locality [15, 31], effective graph data cache management that can exploit this serialization to improve hit rates is an open research problem.

In this paper, we propose a new cache replacement policy for disk-based GDBMSs. Our proposal, called *Locality Aware Caching* (LAC), leverages the graph structure to estimate the most beneficial pages to keep in the cache, thus minimizing the number of page faults by improving graph data locality. For a series of read requests, LAC's objective is the same as in any cache management technique: to fetch disk pages on demand into the cache and evict pages from the cache such that the number of cache misses is minimized. LAC's approach leverages graph access patterns: *read operations on graphs tend to explore local neighborhoods of a vertex first and then expand the neighbourhoods in multiple hops through repeated access patterns.* Therefore, the spatial locality that is captured in disk pages also offers temporal locality for the subsequent accesses of cached pages. LAC exploits this property in deciding which pages to retain in the cache when making eviction decisions.

The contributions of this paper are the following:

- We address the challenges in identifying correlations between data accesses and the topology of graphs to increase the access locality.
- We propose LAC, Locality-Aware Cache Replacement Policy that increases cache utilization by retaining pages in the cache that are likely to be accessed together.
- We evaluate LAC against the well-known and widely used cache replacement policy GCLOCK on input graphs with different structural properties while running various query types using both simulations and a real system integration, and show that LAC achieves improvements by up to 1.42× in simulations and 1.23× in the real system for query execution time.
- We evaluate LAC against a graph topology-aware cache hit promotion policy, GRASP, on the vertex pages of input graphs while running various query types using simulations. We show that LAC can outperform GRASP in most cases in page fault and latency.

2 BACKGROUND AND RELATED WORK

We split this section into two parts. First, we introduce background information about topology-aware graph layout generation algorithms and their impact on improving the performance of cache management. Second, we highlight studies in the literature and practical solutions in commercial systems that employ various cache management techniques while running graph applications.

2.1 Background

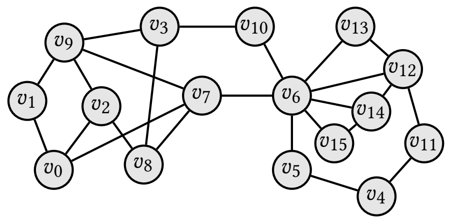
It is beneficial to reduce the number of disk page accesses to answer a query by exploiting data access locality. In graph traversals, these accesses are correlated with graph topology. Thus, if data are stored in a page layout that is correlated with the graph topology, then the number of pages that are accessed during traversals would be reduced. If data in this layout are also retained in data caches, this would improve the cache hit rate and thereby minimize page faults required to access data on disk. Given that cache space is limited, there is a need for cache replacement policies that are aware of graph data access patterns so that the cache hit rate can be improved, thereby improving performance.

A number of techniques have been proposed to optimize the locality of accesses in storage layers [7, 9, 15, 29, 30, 34]. Some of these techniques use expensive graph partitioning and community extraction approaches for determining the serialization layout [17, 23]. There are also studies that order the graph vertices according to their structural properties by relabelling the vertex identifiers and serializing the input graphs. The reordering approaches are based on the fact that most GDBMSs ingest input files with a given vertex order and load them into disk pages. Hence, ordering vertices based on topology ensures that vertices that are likely to be accessed close to each other are placed on the same or on consecutive pages.[3] Some of the ordering approaches exploit the skewed distribution of vertex degrees – a well-known structural property – that appears in most of the real-world graphs [9, 34]. Placing higher degree vertices together on the same page increases the reuse likelihood of those pages and helps in making better replacement decisions.

Graph topology-aware ordering and layout generation on disk can help caching in three ways:

(1) It exploits intra-page spatial locality. Increased locality reduces page accesses. The application requires fewer distinct pages to answer a query which results in less IO and therefore, fewer page faults.

[3]Some of the example graphs that are available in public repositories are already ordered this way, but not all of them are. Consequently, we have observed that they have highly variable performance based on the assignment of vertex identifiers in the input files.



(a) Topology of an example input graph.

Order	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v_i	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
	Page 1				Page 2				Page 3				Page 4			

(b) Vertices are laid out in pages according to their identifiers given in the default input file.

Order	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v_i	v_6	v_3	v_9	v_8	v_7	v_2	v_0	v_1	v_{10}	v_5	v_4	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
	Page 1				Page 2				Page 3				Page 4			

(c) Vertices are laid out in pages according to their identifiers given by a topology-aware graph ordering approach [29].

Figure 3: Different disk layouts for input graph vertex pages.

(2) It exploits inter-page spatial locality. Increased locality enables cache replacement policies to make better decisions by following the successive access likelihood of pages.
(3) It gathers hub graph vertices/edges in the same pages and therefore increases the reuse likelihood of those pages whose access likelihood follows the skewed degree distribution of vertices.

Figure 3a shows an example undirected input graph with assigned identifiers. Figure 3b shows vertex pages on disk (edge pages are excluded to simplify the example) when the vertices are laid out sorted by their identifiers and the page capacity is four. When the graph application explores immediate neighbourhood (common in BFS-based algorithms) of vertex v_7 that has four neighbouring vertices (v_0, v_6, v_8, v_9), it requires access to three distinct pages to execute the query: $\{p_1, p_2, p_3\}$.

Figure 3c shows an alternative layout for the vertex pages of the same input graph which is ordered by a topology-aware graph ordering algorithm [29]. When the immediate neighbourhood of vertex v_7 are explored in this case, we see that the application accesses only two distinct pages, $\{p_1, p_2\}$. When we run the same query for all vertices, we observe that the layout in Figure 3c reduces the number of page accesses by 14%. We also observe the increased reuse likelihood of some pages with hub graph vertices. When the immediate neighbourhood query is run for all vertices, the application on the layout in Figure 3c performs 40% of its page accesses on a single page, which is 35% of the accesses in the default layout in Figure 3b. On real large graphs, the improvement is higher.

The layout in Figure 3c is obtained by the state-of-the-art graph ordering approach, Gorder [29]. Gorder determines the access locality of vertex pairs by explicitly defining a score function (*gscore*) that considers the immediate neighbourhood of each vertex and the number of common friends between two vertices. It serializes to maximize *gscore* within a window of a predefined length w which slides over the vertices in the default order of the graph in the input file. The score function is defined on every pair (u, v) of vertices within the window as follows: $gscore(u, v) = a_{uv} + |N(u) \cap N(v)|$, where $a_{uv} = 1$ if there is an edge between vertices u and v (assume an undirected graph) and 0 otherwise, and $N(u)$ ($N(v)$) is the number of immediate neighbour vertices of u (v). *gscore* represents how well the serialization captures immediate neighbourhood of vertices and neighbourhood overlap within a page.

Gorder is designed for in-memory graph processing, and therefore targets improving the performance of CPU caches and sets the default window size w to be the size of a CPU cache line. In our work, we have adapted Gorder to work on secondary storage serialization and to use improved locality on disk pages in designing our replacement policy.

2.2 Cache Management

As in all disk-based systems, GDBMSs rely on memory cache layers for performance improvements. These systems mostly use replacement policies based on recency and frequency without taking into consideration the graph topology. Most GDBMSs typically use existing replacement policies based on spatial and/or temporal locality. Some use the graph ordering approaches discussed above.

Among general-purpose cache replacement approaches, Least Recently Used (LRU) is perhaps the most popular [6, 13, 20]. LRU relies on the observation that recently used pages are more likely to be re-accessed in the near future. Every time a page is accessed, LRU replacement policy updates its recency by placing it in the most recent position. When there is a need for page replacement, LRU evicts the page that is in the least recent position. LRU's reordering in cache is costly, which has led to other approaches that approximate it. One popular alternative is CLOCK, which avoids costly reordering and captures the recency of accesses by arranging cache pages in a circular list and keeping a reference bit on each page. The reference bit is set every time a page is accessed. When a page needs to be evicted, CLOCK replacement policy scans the pages in the circular list and checks their reference bit. If the reference bit on a page is set, CLOCK unsets it and moves to the next page. The first page with an unset bit is replaced. There are recently published and well-performing graph database engines [11, 12, 28] that use CLOCK as a page replacement policy. However, neither LRU nor CLOCK consider access frequency. Therefore, other GDBMSs such as Neo4j [1] instead use the Generalized CLOCK (GCLOCK) page replacement algorithm [27]. GCLOCK is an approximation of LRU-K policy [21], which evicts the page whose K-th most recent access is furthest in the past, therefore it considers both recency and frequency of accesses. GCLOCK maintains a reference counter (instead of a reference bit) on each page that is incremented by 1 at each access up to a user-defined upper bound. GCLOCK scans a circular list, decreases the reference count of scanned pages by 1, and evicts a page if its reference count reaches 0. These popular replacement policies are designed to predict future data accesses that consider the spatial and/or temporal locality that exists in the physical data order in cache hierarchies.

There are proposals for cache management techniques that exploit the graph ordering approaches discussed in the previous subsection. CAGRA [34] targets in-memory graph processing systems and assumes that the graph is serialized in memory by using frequency-based clustering [34]. It maintains the cache content by partitioning the data in the cache to reduce random memory accesses. It introduces CSR-segmenting which splits the set of vertices into cache-line size segments, and creates subgraphs in each segment by placing destination vertices and edges to those destination vertices. GRASP [10] introduces cache insertion and hit-promotion cache policies for in-memory graph processing systems to increase the reuse likelihood of last-level-cache blocks. GRASP assumes that the graph is serialized by using DBG [9] vertex ordering algorithm which is designed to exploit vertex degree distribution of graphs and partition vertices into a small number of groups based on their degree. Cache pages are grouped according to their usage levels: high, moderate, and low. Pages are fetched on demand, but if they are in the high-reuse group, they are inserted at the most recently used position in the cache. Moderate and low-reuse groups are inserted near or at the least recently used position to make them immediate candidates for eviction decisions. GRASP also introduces immediate or gradual hit promotions for different reuse groups to maintain their residence in the cache. GRASP designs the position of pages in the cache and uses RRIP [16] for replacement decisions. Zhou et al [37] propose a batch replacement policy to avoid random IO accesses in buffer managers of update-intensive GDBMSs. It relies on continuous serialization of neighbouring communities on disk and search the longest continuous sequence of pages to find the eviction candidate pages as a batch. It does not use access frequency in replacement decisions. Clock-based graph-aware caching (CBGA) [2] proposes a graph topology-aware replacement policy to evict graph objects from a cache layer that is located between distributed graph storage and processing node. Once a graph object is fetched into the cache, it is assigned a cost value that considers network latency to fetch it and its hop distance to the vertex that is being queried. When the cache is full, a similar approach to GCLOCK policy is used, and graph objects whose cost value reaches zero are evicted.

In this work, we show that when the locality of access is improved in the storage layer, instead of using generic replacement policies based only on recency and frequency information, a replacement policy that takes into account graph topology in addition to recency and frequency improves the performance of memory cache layers in disk-based graph processing environments.

3 GRAPH TOPOLOGY-AWARE CACHE MANAGEMENT

The absence of locality in graph applications makes cache replacement decisions challenging. Figure 2 shows that locality can be imposed through serialization on disk induced by graph topology-aware ordering algorithms. The improved locality on disk enables better cache replacement policies with awareness of graph data access patterns.

In this section, we propose a locality-aware cache replacement policy called LAC. LAC is sensitive to graph topology in reducing the number of page faults while servicing user requests. LAC's approach considers that the spatial locality that is captured in properly serialized disk pages also offers temporal locality for the subsequent accesses of cache pages.

3.1 LAC Overview

A graph $G = (V, E)$ consists of set of vertices V and edges $E = \{(u, v) | u, v \in V\}$. The number of edges linked to a given vertex v is its *degree*, $d(v)$. The on-disk serialization of a graph G (denoted G_D) consists of a set of k pages: $G_D = \{p_0, p_1, ..., p_{k-1}\}$, where i is the identifier assigned to page p_i during serialization, and $k = \lceil \frac{|V|+|E|}{|p_i|} \rceil$ is the number of disk pages. We formulate a cache, $G_C \subseteq G_D$ consisting of s pages ($s \leq k$) that are currently in the cache.

When an application accesses page p_{in}, LAC checks if p_{in} is already in the cache. If it is, then a cache hit occurs, and LAC does not need to do anything (returns null). Otherwise, a cache miss occurs. If G_C is already full, LAC's page eviction procedure finds the page to be evicted from the cache, p_{out}, and p_{in} is fetched into G_C from G_D. This is common to all cache management algorithms; where LAC differs is in its page eviction procedure.

LAC's page eviction solves an optimization problem that minimizes LAC's cost functions to maintain the cache content in terms of graph-aware spatial and temporal locality of cache pages. The procedure finds the page to be evicted, p_{out}, by capturing the reuse likelihood of pages and considering the access locality of pages that are laid out on disk by using a locality-aware ordering algorithm. Sections 3.2 to 3.4 explain these in detail.

3.2 LAC's Cost Functions

As pages are brought into the cache on demand and the cache space is managed, the objective is to make sure that the cache is as beneficial as possible in minimizing page faults. In LAC's case, this means that pages that represent communities in the graph are retained in the cache, paying attention to their expectation of being accessed in the near future.

When a topology-aware graph ordering algorithm serializes a graph on disk pages, its aim is to place neighbouring communities in the graph on the same page and, if needed, on consecutive pages. Therefore, vertices and edges that are located, for example, on pages p_i and p_{i+1} have stronger neighbourhood connection than those on pages p_i and p_{i+10}. The ordering techniques, as discussed, differ in how they identify communities. LAC exploits this property by computing neighbourhood distances using page identifiers.

Let us assume, for simplicity, that the cache G_C contains the first s pages $p_0, \ldots, p_{s-1}$ where the subscript indicates the page identifier. Then $G_C[1] = p_0, \ldots, G_C[s] = p_{s-1}$ where $G_C[i]$ indicates the $i-$th position in cache. For each page p_i in G_C, its *prev* and *next* pages are defined; these are kept in a cache metadata structure:

$$\begin{aligned} prev(G_C[i]) &= p_j \text{ such that } argmin_j(i-j) > 0, \forall p_j \in G_C \\ next(G_C[i]) &= p_j \text{ such that } argmax_j(i-j) < 0, \forall p_j \in G_C. \end{aligned} \quad (1)$$

The edge conditions are $prev(G_C[0]) = next(G_C[s]) = p_\infty$.

In the remainder, for simplicity, we will abuse notation and use $prev(p_i)$ to mean "the identifier of the page that is located immediately before p_i on disk which is also fetched into the cache" (similarly for $next(p_i)$).

Consequently, the graph topology-aware serialization of the vertices on disk pages represent their likelihood of being accessed together. We define $dist(p_i, p_j) = |i - j|$. Then the *Distance Cost Function* (DCF) for page p_i is defined as:

$$DCF(p_i) = \min\{dist(p_i, prev(p_i)), dist(p_i, next(p_i))\}. \quad (2)$$

$DCF(p_i)$ calculates the page in the cache that is closest to p_i to capture affinity between vertices on two pages. For example, if $dist(p_i, prev(p_i)) = 1$ but $dist(p_i, next(p_i)) = 1000$, it is likely that $next(p_i)$ contains vertices that belong to another community, but there are still vertices in $prev(p_i)$ with which vertices in p_i have an affinity.

We aggregate these to define DCF for the entire cache, G_C:

$$DCF(G_C) = \sum_{p_i \in G_C} DCF(p_i) \quad (3)$$

$DCF(G_C)$ provides a measure of the spatial locality of the vertices as serialized on pages. One objective of LAC, when it is making decisions regarding cache page replacements, is to find a cache page allocation that minimizes $DCF(G_C)$. When a page is fetched into or hit in the cache, its neighbouring disk pages are likely to be accessed in the subsequent requests as a result of the traversal access pattern in graphs. Therefore, minimizing $DCF(G_C)$ exploits spatial locality in the cache that is captured on disk.

Temporal locality is another important aspect to increase the reuse likelihood of cached data. While the replacement policy maintains the closeness of pages residing in the cache, it should also make room for recently accessed pages that can be requested as a result of accessing a new community in the graph (as a consequence of the application's working set). However, the recency of references cannot capture page access frequency. Therefore, it is important to consider infrequent page references during the eviction decision. In graph applications, immediate neighbourhood exploration of a vertex usually happens once. However, due to common skewed degree distributions, hub vertices and their immediate neighbourhoods are likely to be frequently accessed since they are usually involved in multiple neighbourhoods. Therefore, it is common for data pages to suddenly become sufficiently popular to keep in the cache or sufficiently unpopular to be evicted from the cache.

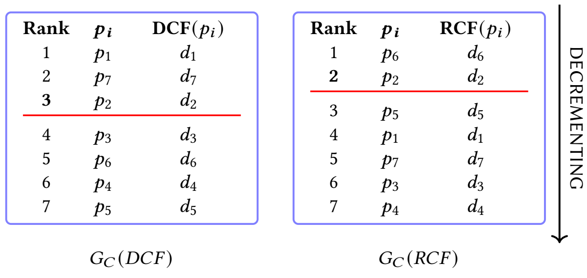
Frequency of page accesses with a notion of aging is widely used in popular cache replacement policies such as LRU-K [21] and GCLOCK [27]. LAC uses this same notion and takes into account access history by considering the time of the last K references of each page p_i in G_C, R_{p_i}, where $K \geq 1$: $R_{p_i} = \{R^1_{p_i}, R^2_{p_i}, ..., R^K_{p_i}\}$.

LAC maintains for each page a *Reference Time Cost Function*, RCF(p_i) that considers the K^{th} most recent reference of a page to adapt the cache content to recent and frequent access pattern changes during traversals. When a page p_i is first fetched into the cache or gets a hit in the cache, R_{p_i} is updated. Given a value R_{p_i} known up to the current clock value t_{now}, the reference time cost of p_i is defined as $RCF(p_i) = t_{now} - R^K_{p_i}$.

Accordingly, a *Reference Time Cost Function* of the entire cache, $RCF(G_C)$ at time t_{now}, is defined as follows:

$$RCF(G_C) = \sum_{p_i \in G_C} RCF(p_i). \quad (4)$$

Relying on the reuse likelihood of recently and frequently accessed cached data, minimizing RCF(G_C) exploits the temporal locality that is also offered by the spatial locality in disk pages captured by an ordering algorithm.



Rank	p_i	DCF(p_i)
1	p_1	d_1
2	p_7	d_7
3	p_2	d_2
4	p_3	d_3
5	p_6	d_6
6	p_4	d_4
7	p_5	d_5

$G_C(DCF)$

Rank	p_i	RCF(p_i)
1	p_6	d_6
2	p_2	d_2
3	p_5	d_5
4	p_1	d_1
5	p_7	d_7
6	p_3	d_3
7	p_4	d_4

$G_C(RCF)$

Figure 4: Rank-based Pareto filtering approach.

3.3 Cache Optimization Formulation and Possible Solutions

We formulate LAC's page eviction decision as a bi-objective optimization problem that minimizes *DCF* and *RCF* of the cache. The constraints of the optimization problem bound the values of *DCF* and *RCF*.

$$\begin{aligned} &\min \quad (DCF(G_C), RCF(G_C)) \\ &s.t. \\ &s + 1 \leq DCF(G_C) \leq k \\ &0 \leq RCF(G_C) \end{aligned} \quad (5)$$

$DCF(p_i)$ values range from 1 to $k-1$, where k is the total number of disk pages; thus $DCF(G_C)$ ranges from $s+1$ to k where s is the cache size in the number of pages. However, there is no natural upper-bound for *RCF*; its lower bound is 0.

The bi-objective optimization can be scalarized to a single objective formulation. The simplest case is weighted sum scalarization. Applying this technique to our case requires normalization of DCF and RCF to ensure that their contributions are properly accounted for and to avoid introducing extreme bias toward either objective. However, as noted, RCF has no natural upper bound, making normalization infeasible. An alternative would be to use a rank-based Pareto filtering method [5] to find one solution.

The Pareto filtering method can be implemented using a rank-based skyline operator [5] in order to filter out the non-dominated pages from replacement consideration. The algorithm would maintain two sorted lists of the pages in the cache (in G_C): G^{DCF}_C that is sorted in descending order of *DCF* and G^{RCF}_C sorted in descending order of *RCF* as shown in Figure 4.

When a page p_{in} is fetched and the cache is full, requiring a page to be evicted, these lists are used in the following manner. Page p_{in} is inserted in both lists in its appropriate place. Then the first common page that appears in both $G_C(DCF)$ and $G_C(RCF)$ lists is selected (p_{out}) as one of the Pareto solutions for eviction. In Figure 4, p_2 is the candidate page for eviction.

This method would be reasonably efficient, since it would not find the entire Pareto front and stops when it finds the first candidate as this would be sufficient for our purpose. However, the algorithm is still linear to the cache size and it requires maintaining two sorted lists. Furthermore, the scan time of pages to find the first common page can be costly, especially during the traversal from one graph community to another. These drawbacks make this solution expensive as also shown in our experiments in Section 5.2. Therefore, we introduce LAC as a lightweight heuristic that solves the overhead and normalization problems in the online solution. We describe LAC next.

3.4 A Lightweight Heuristic for Page Eviction

LAC is a lightweight heuristic algorithm for solving this optimization problem based on GCLOCK [27], which is a widely-used cache replacement algorithm that takes access history into account. As a quick recap, GCLOCK maintains a circular list for the pages in the cache and each page is associated with a reference counter, RC_{p_i} which is updated as explained in Section 2. GCLOCK can be used to keep track of the access history of cache pages in order to minimize RCF defined in LAC's cost functions.

Algorithms 1 and 2 show the main steps of LAC in maintaining page fetches and evictions in the cache.

Algorithm 1 LAC: Locality-Aware Cache Replacement

```
Input: $p_{in}$: page to be fetched from $G_D$
Output: $p_{out}$: page to be evicted from $G_C$
1:  $G_C^{circular}$: a circular list for cache pages
2:  if $p_{in} \in G_C$ then                       ▷ A cache hit occurs
3:      IncrementRC($p_{in}$)   ▷ Increment reference counter of $p_i$
4:      $p_{out} \leftarrow NIL$
5:  else                                          ▷ A cache miss occurs
6:      if $|G_C| > s$ then
7:          $p_{out} \leftarrow$ PageEviction-Lac($G_C^{circular}$)
8:          $G_C \leftarrow G_C \setminus \{p_{out}\}$
9:      end if
10:     IncrementRC($p_{in}$)
11:     $G_C \leftarrow G_C \cup p_{in}$
12: end if
```

Algorithm 2 PageEviction-Lac

```
Input: $G_C^{circular}$: a circular list for cache pages
Output: $p_{out}$: page to be evicted
1:  $clock \leftarrow 0$
2:  do
3:      $p_c \leftarrow G_C^{circular}[clock]$
4:      if $RC_{p_c}$ is 0 then
5:          $p_{out} \leftarrow p_c$
6:      else
7:          $RC_{p_c} \leftarrow$ DecrementRC($p_c$)
8:      end if
9:      clock += 1
10: while $p_{out}$ not found
```

LAC considers DCF to exploit spatial locality among the graph objects during traversals. Therefore, modifications are needed in the handling of the promotions (demotions) of cache pages.

LAC keeps track of a counter set-value for each p_i, (SET_{p_i}) that is based on the current state of p_i, $SET_{p_i} = DCF(p_i) \div MaxDist$, where $MaxDist = k{-}s$ is an upperbound for $DCF(p_{in})$ derived from Equation 2. Therefore, each p_i is promoted or demoted according to the strength of its connection to its neighbouring community residing in the cache. Since $DCF(G_C)$ is to be minimized, a page p_i's contribution to RC_{p_i} needs to be large, if $DCF(p_i)$ is small. Therefore, RC_{p_i} can prevent p_i from early evictions. For that reason, normalization by $MaxDist$ is performed, and once p_i is fetched or hit in the cache, promotion is done by $(1 - SET_{p_i})$ until it reaches the value of user-defined parameter K. Algorithm 3 shows the promotion of a page that is recently fetched into or hit in the cache.

Algorithm 3 IncrementRC

```
Input: $p_{in}$: page to be hit or fetched
Input: $K$: farthest last reference of a page to keep track of
Output: $RC_{p_{in}}$: reference counter of $p_{in}$
1: if $RC_{p_{in}} + (1 - SET_{p_{in}}) \leq K$ then
2:     $RC_{p_{in}} \leftarrow RC_{p_{in}} + (1 - SET_{p_{in}})$
3: end if
```

On the other hand, a page p_i's demotion reflected in RC_{p_i} needs to be small if $DCF(p_i)$ is small, and large if $DCF(p_i)$ is large. This is because if $DCF(p_i)$ is large, it means that its strength of its connection to its neighbouring community in the cache is weak. Therefore, it is a good candidate for eviction, and demotion needs to consider this. Algorithm 4 shows the demotion of a page pointed by clock during the replacement decision.

Algorithm 4 DecrementRC

```
Input: $p_c$: page to be pointed by clock
Output: $RC_{p_c}$: reference counter of $p_c$
1: $RC_{p_{in}} \leftarrow RC_{p_{in}} - SET_{p_{in}}$
```

Calculating SET_{p_i} can be achieved in constant time and the implementation details are presented in Section 4.2. Therefore, the worst-case time complexity of LAC is linear to the cache size. However, as shown for GCLOCK [19], the iteration over the pages in the cache for finding the one with zero reference count can converge quicker in practice depending on the workload access pattern.

This paper focuses on read-only workloads. LAC uses distance computations to update the reference counters associated with cache pages. To perform this task efficiently, it maintains an in-memory data structure that captures the layout and number of disk pages with other metadata (Section 3.2). If graph updates were to be supported, then this metadata would require updating to maintain accuracy, and its maintenance can be done incrementally.

4 EXPERIMENTAL SETUP

We evaluate LAC both in a simulated setup and embedded in a real GDBMS. We present our setup, experiments and results in subsequent sections.

4.1 Simulated Setup

The simulated setup used in the evaluation is depicted in Figure 5. We simulate direct IO to fetch pages to in-memory cache from the disk. The simulated setup gives us full control of the environment to isolate our tests.

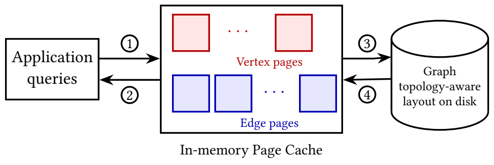


Figure 5: Simulated three-layer system architecture where an input graph laid out on disk.

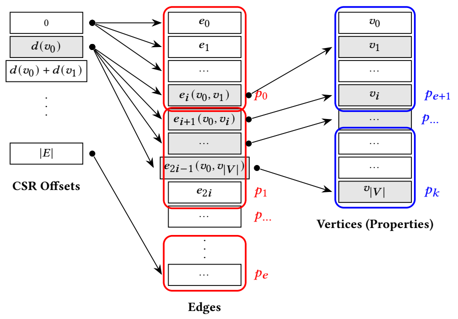


Figure 6: Input graphs' serialization layout on disk.

The in-memory cache stores both vertex and edge pages. The workflow is typical: $\textcircled{1}$ applications access data pages in the cache; $\textcircled{2}$ if a page is in the cache, it is returned and this counts as a cache hit; $\textcircled{3,4}$ if the page is not in the cache, it is fetched from secondary storage and counts as a cache miss. LAC manages the in-memory cache.

Input graphs are ordered using the Gorder algorithm.[4] The graph datasets are serialized into disk pages using Compact Sparse Row (CSR) format as shown in Figure 6. CSR Offsets array has $|V|$ entries and each entry determines the alignment of the corresponding vertex's neighbourhood in the Edges array which is split into edge pages. Vertices and their properties are stored in the Vertices array which is split into vertex pages. When a query starts with a source vertex, its neighbouring vertices are found by following the offsets in the Edges array, and depending on the application, destination vertices' properties are visited in the Vertices array. We follow the data layout in [22] where each vertex page can store 512 vertices and each edge page can store 1024 edges.

In order to determine the IO latency in the simulation, page size is set to 4KB.[5] A database file is created on disk aligned with the page size, and pages are laid out in the file following the topology-aware layout generation obtained by Gorder. The average latency for each page IO is measured as 150 microseconds on a system that has 400 GB Intel S3700 SSD and this is the value used in the experiments to calculate query latency.

4.2 Real System Integration

LAC can be implemented in GDBMSs that have a page cache module [1], or in a file IO interface [35], and can be used in out-of-core graph processing systems [36]. In this work, we integrate LAC into Neo4j GDBMS [1] to evaluate its applicability and efficiency in a real system.

Neo4j treats vertices, edges and properties separately. Instances of each are stored as disk records organized into separate files, i.e., node files, relationship files. Records of each file type have identical sizes, but they differ among different file types. Records of a given file type have unique identifiers. Neo4j uses a shared page cache for all record types. The page cache is managed by GCLOCK replacement policy. Each cache page mapped to a disk page has a reference counter to be incremented or decremented when there is a hit or miss in the cache, respectively, as discussed previously.

Using Neo4j's GCLOCK implementation as a starting point, we implement Algorithms 4 and 3 and update the reference counters of cached pages according to LAC's methodology. Neo4j has other record types for auxiliary data beyond the vertex, edge and properties and these are also cached. LAC manages and maintains only the vertex, edge, and property pages within the page cache. The remaining store types are left to Neo4j to manage. This design choice is necessitated by the fact that LAC accounts for the topology-awareness of the disk layout, which can only be effectively ensured for edges and vertices by the graph ordering.

The calculation of SET_{p_i} for each page is triggered either upon a hit on the page or when the clock arm traverses the cached pages in search of an eviction candidate. To satisfy the constraints of Equation 1, it is necessary to maintain sorted metadata for the identifiers of cached pages. However, as sorting operations are computationally expensive for updates, a trade-off is made by utilizing a small amount of additional memory (0.05% per 1 GB of disk data) to store float arrays. These arrays, sized according to the number of pages on disk per file type, record information about pages currently in the cache along with their reference counters and are updated using LAC's methodology. Using these arrays, the value of $DCF(p_i)$ can be determined by a lookup that identifies the nearest preceding and succeeding pages in the cache. This lookup requires at most $MaxDist$ iterations on either side, as described in Section 3.4, where $DCF(p_i)$ is normalized. Furthermore, Section 4.6 establishes that $MaxDist$ has a practical upper bound, which is a constant (typically around 5). Consequently, the update operation for reference counters achieves O(1) complexity.

We evaluate the performance of LAC in comparison to GCLOCK using an embedded Neo4j database. The Java Traversal API was utilized to implement the queries, enabling direct control over the traversal logic and allowing customization tailored to specific graph structures. We import databases into Neo4j by setting `id-type` as `ACTUAL` and using the default `aligned` record format. We set `direct-io` to `true` in Neo4j to ensure that pages are fetched from disk directly into Neo4j's page cache. We also disable the background eviction thread in the page cache to control the insertion and eviction of pages as we do in our simulated setup.

4.3 Datasets

The premise of LAC is that after proper serialization to provide locality, a page is likely to be successively accessed if it contains neighbouring vertices or edges that are reachable within a couple of hops. This is due to the neighbourhood expansion access pattern of vertices. However, structural properties of different graphs, such

[4]We set the window parameter in Gorder as 512 since each vertex page consists of 512 vertices in our simulation.
[5]This is a tunable parameter in the simulation.

Table 1: Input graphs for performance evaluation of GAL

Graphs	$\|V\|$	$\|E\|$	Diameter
AMAZON	0.4M	3.3M	7.6
PATENT-CIT	3.7M	16.5M	9.4
FLICKR	2.3M	33.1M	6.8
SOC-LJ	4.8M	68.4M	6.1
TWITTER	41.6M	1.4B	3.9

as degree, change the reuse likelihood of pages and affect the performance of any replacement policy. Therefore, the experiments are performed over a set of input graphs with different properties as shown in Table 1

SOC-LJ[6], FLICKR[7] AND TWITTER[8] are mid- and large-size online social networks where vertices and edges represent the friends and friendship relations, respectively. Online social networks are dense graphs with small diameters, meaning that vertices are reachable within a small number of hops. PATENT-CIT[9] and AMAZON[10] are web graphs whose vertices and edges represent web pages and hyperlinks between them, respectively. The average degree of web graphs is slightly smaller than online social networks, however their diameters are larger.

4.4 Workload

In this study, we primarily focus on different types of traversal queries where only some parts of the input graphs are traversed starting from user-defined vertex/vertices. Query set in the workload is influenced by widely-used LDBC Social Network Benchmark (SNB) [8]. The LDBC SNB query workload is designed to work on heterogeneous (synthetic) graphs, but the input graphs used in this study (Table 1) have homogeneous vertices and edges. Therefore, we modify the SNB queries to work on homogeneous (real) graphs.

4.4.1 Immediate Friends. Accessing the immediate neighbourhood is a common workload in most graph applications from different domains. It is the building block of path finding, reachability and even point queries whose complexities are sublinear to the dataset size and are widely used in social network benchmarks [8]. **FR-ALL** accepts a vertex u and traverses its immediate edges and vertices. Accessing immediate neighbourhood can be accompanied by different filters. Many examples can be found in LDBC SNB benchmark, such as friends, posts or comments with different properties within the 1-hop distance of a given start vertex. In this workload, we use the degree property of vertices to filter FR-ALL which is used to represent the hotness of vertices in graphs, meaning that a vertex with higher number of immediate neighbours has a higher degree. **FR-HOT** query accepts a vertex u along with a threshold parameter d and traverses the immediate edges and vertices whose degree is greater than or equal to d.

4.4.2 Reachability. Exploring beyond the immediate neighbourhood is another important access behaviour of graph applications, e.g., to find friends with mutual interest or explore the reachability paths among the graph objects. These applications usually require breadth-first traversal access pattern within a maximum depth. Reachability queries can explore different parts of the input graphs at the same time. Therefore, the page accesses can be more scattered than the immediate neighbourhood explorations. We choose to evaluate a single pair shortest path query (**SPSP**) which finds the shortest path between two vertices u and v. In this study, we use a version of this workload that restricts the (hop) distance between two vertices to r; if they are further apart, the query result is empty (path is not found). We implement SPSP query by performing a bi-directional search [24].

4.4.3 Random Walk. Random Walk (**RW**) is a widely used path finding query among many different graph applications. In contrast to immediate neighbourhood, it explores the connections along various communities in a graph. RW does this exploration by performing a number of walks (NW) starting from a vertex and a number of steps (NS) to hop between the neighbourhoods. Therefore, a RW query accepts a vertex u and NS and performs NW random walks that start at u, and returns the vertices and edges that are accessed along the traversed paths.

4.4.4 Mixed Workload. Finally, we design a mixed workload (**MIX**) to evaluate LAC's adaptability to changes in the access patterns. This workload runs a mixture of query types with different traversal patterns and depths. For each query execution, a query type is uniformly randomly selected along with the necessary input parameters while guaranteeing that corresponding experimental set-ups have the same random set of query types and parameters.

4.5 Cache Replacement Policies

In previous work [19], it has been shown that GCLOCK performs better than LRU in terms of page hits under certain conditions. It also has lower computational overhead relative to LRU. GCLOCK is also employed in the page cache layer for edge and vertex pages of modern GDBMSs such as Neo4j [1]. Therefore, we compare LAC against GCLOCK in both simulated setup and real system integration.

As mentioned in Section 2, GRASP is a graph structure-aware hit and fetch promotion policy used in CPU caches of in-memory graph processing systems while running analytical queries. Its policies are designed to reduce the random memory accesses on vertex (property) arrays. GRASP is also layout-sensitive and assumes that the input graph is ordered by using Degree-based Grouping (DBG) algorithm. DBG creates multiple bins for vertices and places them into these bins according to their *hotness level*, which is again defined based on their degrees. Vertices in each bin are ordered according to their order in the input file. We implement GRASP's promotion policies and use GCLOCK for replacement decisions, and compare it with LAC on vertex pages in simulated setup.

4.6 Evaluation Metrics

In this paper, we primarily focus on **query latency** as the metric. Query latency comprises **IO time** (affected by page faults) and replacement policy execution time (**policy overhead**). The IO time is affected by the number of **page faults** and **IO latency**. We also focus on comparative values, i.e., LAC's performance against GCLOCK and native Neo4j rather than absolute values.

[6]http://konect.cc/networks/soc-LiveJournal1/
[7]http://konect.cc/networks/flickr-growth/
[8]http://konect.cc/networks/twitter/
[9]http://konect.cc/networks/patentcite/
[10]http://konect.cc/networks/amazon0505/

In the simulation study, we examined the page faults of each algorithm. We refer to these experiments when necessary, however, we do not explicitly report them due to space considerations. In the simulation study we compute IO time by multiplying the number of page faults with IO latency per page. Our measurements show that LAC's policy overhead is negligible compared to IO time. Thus, query latency correlates well with the IO time.

In Neo4j integration, we measure the time it takes to execute a set of queries including the actual IO time by performing direct IO.

5 EXPERIMENT RESULTS

This section presents the experimental evaluation of LAC in latency improvements over existing replacement (GCLOCK) and hit/fetch promotion (GRASP) policies. We first explain the parameter settings for the workload design and algorithm initializations. Then, we show and discuss the experimental results both in the simulated setup and the real system.

5.1 Parameter Settings

5.1.1 Query Initialization Parameters. The workload size for FR-ALL, FR-HOT and RW is set to 50,000. It is set to 5,000 for SPSP. In Neo4j experiments, due to the size and skewness of large graphs, we set the workload size to 2,000 for SPSP query. For SPSP queries, r is set to three for social networks [3] following the degree of separation between two vertices. On the other hand, r is set to the effective diameter of web graphs. Similarly, for RW queries, NW is set to the average degree and NS is set to the effective diameter of the input graph.

5.1.2 LAC Initialization Parameters. Recall from Section 3 that LAC uses two parameters. One parameter is K, which represents the farthest last reference of a page (Section 3.4). K is also the only parameter for GCLOCK. Setting $K = 4$ by default is considered to be performant in Neo4j [1]. Our experiments also confirm this value and we set $K = 4$ for both LAC and GCLOCK. GRASP has different levels for grouping vertices based on their degree to promote vertex pages. We determine four different levels and the maximum value to promote pages is set to 4.

The second LAC parameter is $MaxDist$, which is the upper-bound value for $DCF(p_i)$. Although $MaxDist$ has a theoretical upperbound based on the cache size (Section 3.4), our experiments show a tighter practical bound for this parameter. To determine the practical bound, we run the following experiment. We use SOC-LJ as the input graph and consider SPSP queries on a warm cache whose size is set to 10% of the input graph. We manage the cache using GCLOCK and examine $DCF(p_i)$ for each p_i in G_C. Although for this case, the theoretical upperbound of $MaxDist$ is 180,000, in practice the maximum $DCF(p_i)$ is 438, and 95% of cache pages have $DCF(p)$ values less than 15 as a result of the consecutive page access pattern of queries.

This is important for LAC's performance. Consider pages p_i and p_j with $DCF(p_i) = 200$ and $DCF(p_j) = 2000$, respectively. Since queries in the workload access mostly consecutive pages (as a result of topology-aware serialization on disk), these two pages need to be considered as having similar eviction probabilities. However, if $DCF(p_i) = 1$ and $DCF(p_j) = 5$, p_i is more likely to stay in the cache than p_j. Therefore, $MaxDist$ parameter is set to $(|G_C| \times p)^{th}$ DCF value in G_C^{DCF}, where $0.85 \leq p \leq 0.9$ in order to eliminate the exponential distribution of $DCF(p_i)$ and taking the log (base 10) of $MaxDist$ satisfies this property.

Table 2: LAC's Heuristic Approximation

	Cache Size			
	5%	10%	20%	30%
LAC's page fault approximation	0.90	0.78	0.83	0.89
LAC's policy latency speedup	2.00	2.62	4.38	7.02

5.2 LAC's Heuristic Approximation

As presented in Section 3.4, LAC is a heuristic solution to the optimization problem in Section 3.3 because the online solution (Section 3.3) is too expensive to implement. In this section, we show how closely LAC's heuristic tracks the online algorithm on SOC-LJ when running SPSP queries and K is 1.

Table 2 shows that LAC's lightweight heuristic closely approximates the online solution in the number of page faults. As expected, its policy runtime speedup is up to 7× better, especially when the cache size is large.

We evaluate the page fault performance of the proposed LAC policy in comparison to the optimal cache replacement policy (OPT) [4], using the FR-ALL query workload on the SOC-LJ dataset. In the smallest 5% cache size, LAC achieves a better proximity of approximately 38.72% to OPT compared to GCLOCK that exhibits a proximity of approximately -1.74%. For the largest 30% cache size, both LAC and GCLOCK achieve proximity of approximately 92.75% to OPT. [11]

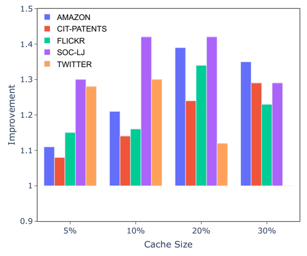
5.3 LAC's Performance in Simulated Setup

Our primary comparison metric is the latency improvements that LAC provides vis a vis GCLOCK and GRASP, which we discuss in this section. We have also evaluated, in detail, the page fault improvements, which is a primary reason we have simulated experiments. We summarize our findings regarding page faults briefly, but primarily focus on latency. The reason is the following.

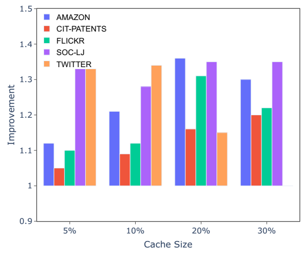
As noted in Section 4.6, two factors contribute to latency: IO latency and policy runtime overhead. As also noted in Section 4.2, LAC's policy runtime overhead is similar to GCLOCK. Therefore, the latency improvements we discuss are primarily driven by IO latency. This latency is linearly proportional to the page fault improvements. Overall, LAC's page fault improvements are up to 1.42× better than GCLOCK. For denser graphs, the improvement is highest when the query accesses vertices in the immediate neighborhood. Conversely, for sparser graphs with larger diameters, the improvement decreases when the query reaches all vertices in the neighborhood beyond the second hop.

5.3.1 Latency Improvements over GCLOCK. Figure 7 shows LAC's latency improvement over GCLOCK in the simulated setup across different cache sizes for the workload over different graphs. Since the cache size is proportional to the graph size, when the cache is

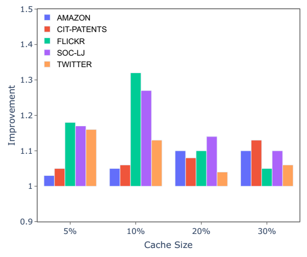
[11] We set a smaller number of workload size than the main experiments and this reduces the impact of differences between the replacement strategies, allowing both policies to exhibit similar performance under larger cache conditions.



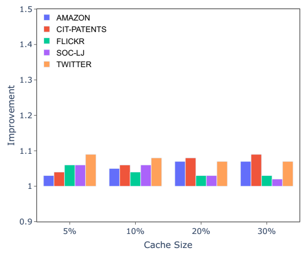
(a) FR-ALL



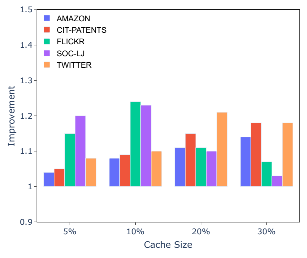
(b) FR-HOT



(c) SPSP

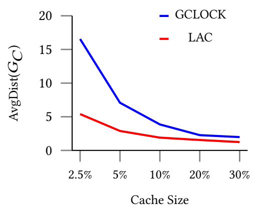


(d) RW

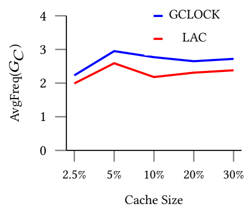


(e) MIX

Figure 7: LAC's latency improvement over GCLOCK in the simulated setup.



(a) $AvgDist(G_C)$



(b) $AvgFreq(G_C)$

Figure 8: LAC vs GCLOCK: Average Distance and Frequency of pages in the cache

large, queries fit into the cache for dense graphs. High skew in degree distribution results in accessing the same pages consecutively. Especially immediate neighbourhood queries do not cause any page faults when the cache is warm. Therefore, no improvement is reported for those cases.

Figure 7a and 7b show that LAC performs better than GCLOCK in all cases. LAC's distance cost function (*DCF*) captures the consecutive access pattern of page accesses within and beyond one-hop traversals better than GCLOCK.

As noted, web graphs have relatively lower average degree than social networks. Therefore, most of the vertices have their immediate neighbouring vertices and edges within at most a few consecutively ordered pages. When the cache size is larger, there can be many distinct communities in the cache, and LAC can better identify and keep them in the cache together. Therefore, as the cache size increases, LAC's improvement over GCLOCK increases in web graphs. LAC outperforms GCLOCK, albeit at a slightly lower level, in FR-HOT queries, especially for social networks. Since social network graphs are highly skewed, frequency has more impact in identifying hot vertices. Since GCLOCK predominantly considers frequency, its performance improves in FR-HOT relative to FR-ALL queries. Therefore, LAC's improvement over GCLOCK is lower for this query type.

Traversing beyond the first hop is another important access pattern in graph applications. Figure 7c and 7d show that LAC outperforms GCLOCK in all cases when the query goes beyond the one-hop neighbourhood in SPSP and RW queries. In social networks, especially denser and highly skewed ones with many closed triangles and vertices with very large degree properties (such as FLICKR and SOC-LJ), it is not trivial to capture access likelihood of pages in large cache sizes, as a single query may span most of the pages. LAC still outperforms GCLOCK, but its improvement is lower compared to FR-ALL and FR-HOT queries.

SPSP queries access most of the disk pages for input graphs with large diameter and/or relatively more skew. Due to the large diameter of web graphs, a single query usually traverses more than 3-hop neighbourhood. The number of pages accessed within the same communities at this depth is usually limited in web graphs when the average degree is small. Therefore, a bidirectional shortest path query reaches distinct communities from both ends. LAC utilizes these properties better if it has enough information to make the decision based on *DCF*. Since these graphs are small, the number of pages in smaller cache sizes is also small; thus, the distinct number of pages from the same community is not sufficient for LAC to make a better decision based on the difference between page identifiers.

As discussed above, frequency is more important in social network graphs. High degree vertices and their corresponding pages are more likely to be accessed frequently. Also, because of their high degree, their immediate neighbourhoods span many pages, and the cache usually contains a few distinct communities with smaller *DCF* values between pages that belong to different communities. This is challenging for LAC's replacement policy, especially as the cache size gets larger, because more pages can reside in the cache, and therefore, their *DCF* values shrink. In this case, the performance difference between LAC and GCLOCK is reduced, but LAC's performance is still better. To explain this better, we define average distance, $AvgDist(G_C) = \sum_{\forall p_i \in G_C} DCF(p_i)/|G_C|$ and average frequency, $AvgFreq(G_C) = \sum_{\forall p_i \in G_C} RC(p_i)/|G_C|$ of pages.

Figure 8 shows $AvgDist(G_C)$ and $AvgFreq(G_C)$ changes for LAC and GCLOCK according to the cache size while running SPSP queries in SOC-LJ. Note that although GCLOCK does not consider distances in replacement decision, we can easily compute *DCF* for pages in a cache whose content is maintained by GCLOCK.

Figure 8a shows that LAC successfully achieves its objective to minimize *DCF* and keeps more neighbouring pages in the cache. However, the gap closes as cache size increases, and therefore, frequency starts to matter more. As Figure 8b shows, although GCLOCK keeps more frequent pages in the cache, LAC keeps up and maintains $AvgFreq(G_C)$ close to GCLOCK, while improving the neighbourhood collocation better than GCLOCK.

In contrast to other query types, RW traverses communities along the path until it reaches a depth that is the effective diameter of the input graph. Therefore, it does not access all neighbouring vertices at a depth, but visits only one and continues onto the next hop. LAC's improvement is smaller than other query types, especially on dense and skewed graphs (e.g., FLICKR), because these graphs have many closed triangles and a random walk from a start vertex mostly moves around those triangles. Therefore, GCLOCK captures triangles that are frequently traversed along the paths.

Recall that the main focus of MIX workload is to evaluate LAC's adaptability as the system switches from one query type to another – our interest is to see if LAC continues to perform similar to the case where only one type of query is running. The query types in MIX workload ensure that the access pattern in the cache changes over time as a result of breadth- and depth-first traversals. Since LAC can adapt to access pattern changes, the performance does not degrade in running the MIX workload.

5.3.2 *Latency Improvements over GRASP.*

In this experiment, we look at how LAC performs on vertex pages in comparison to GRASP. We focus on vertex pages because GRASP works only on vertex pages. Furthermore, vertex pages have special importance and it is important to study how LAC does on those pages. This is because many property graphs have attributes (properties) on vertices and there may be traversal queries with predicates on vertex attributes. Additionally, in topological traversal queries (i.e., those without predicates on attributes), edges are used to guide the traversal to vertex pages from which other edges are followed.

As mentioned earlier, GRASP requires DBG graph serialization algorithm. A key requirement of DBG is that the vertices in the input file are already ordered in a structure-aware manner. This assumption holds for some input graphs, but not all. To test GRASP's sentivity to ordering, we also ran a test where we shuffle the vertex order to lose locality-awareness if it already exists in the input file. We then run SPSP queries and measure the number of disk page accesses. We observe that when the graph's default order already preserves its topology, further serialization using Gorder does not improve page access behaviour. However, the number of disk page accesses more than doubles when the vertex order does not exist in the input file (shuffled case). This is also true in those graphs whose default order does not reflect topology. For these graphs, ordering using Gorder halves the number of disk page accesses.

Consequently, GRASP has highly variable performance based on whether or not the input files are organized to reflect graph topology. DBG ordering does not help in caching with GRASP, when the input graph does not provide any locality in the input file order. Therefore, we first order the graphs using Gorder and then apply DBG ordering, and use GRASP as the hit/promotion policy. Our approach to use the pipeline Gorder $\longrightarrow$ DBG $\longrightarrow$ GRASP enhances GRASP's methodology. We call this setting as **GRASP-GO**. Our experiments show that LAC outperforms GRASP-GO in most cases, however, due to space limitations, we pick a subset of the datasets, and run FR-HOT, SPSP, and RW queries on them to present the results. Table 3 shows LAC's latency improvements[12] over GRASP-GO on vertex pages.

Table 3: LAC's latency improvement over GRASP-GO on vertex (property) pages.

Input Graph	Query Type	Cache Size 5%	10%	20%	30%
CIT-PATENT	**FR-HOT**	1.05	1.03	1.00	0.97
	SPSP	1.16	1.17	1.18	1.20
	RW	1.28	1.32	1.38	1.43
SOC-LJ	**FR-HOT**	1.07	0.98	0.80	0.70
	SPSP	1.11	1.06	0.96	0.92
	RW	1.18	1.16	1.13	1.13

Table 4: LAC's latency improvement over GCLOCK when the graph is serialized on disk by [7]

Input Graph	Query Type	Cache Size 5%	10%	20%	30%
CIT-PATENT	**FR-HOT**	1.07	1.15	1.20	1.19
	SPSP	1.08	1.18	1.29	1.31
	RW	1.09	1.12	1.14	1.15
SOC-LJ	**FR-HOT**	1.38	1.35	1.29	1.23
	SPSP	1.28	1.25	1.13	1.12
	RW	1.12	1.11	1.04	1.02

FR-HOT query only accesses vertices in the immediate neighbourhood whose degree is greater than the average degree of the input graph, and GRASP's policies can easily maintain those vertices and their corresponding pages in the cache when there is enough room. In addition, as explained in Figure 8a, DCF values shrink for social networks when the cache can accommodate a large number of pages. Therefore, when the cache is larger and/or the query is FR-HOT, GRASP-GO outperforms LAC, especially if the input graph is skewed, such as SOC-LJ. Therefore, when DBG ordering algorithm groups those vertices in the same pages and when the cache size is large, GRASP-GO outperforms LAC as expected.

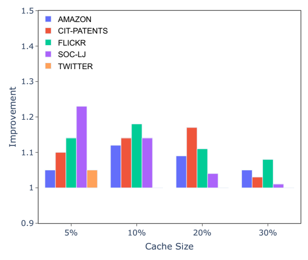
LAC does a better job than GRASP-GO in RW when the query reaches beyond the immediate neighbourhood and does not follow the breadth-first neighbourhood expansion pattern. Random walks bias their exploration towards the degree distribution of the input graphs. Therefore, high-degree vertices are more frequently visited in skewed graphs. Although this behaviour is in favor of GRASP-GO, LAC successfully captures the sudden changes in the access pattern along the path better as random walks can reach distinct communities in the graph.

5.3.3 *LAC's adaptability to ordering algorithms.*

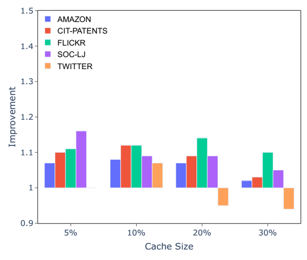
In this section we address the question of how LAC would perform if an ordering algorithm other than Gorder is used. For this experiment, we order SOC-LJ and CIT-PATENT by a new graph ordering heuristic proposed in [7]. We then compare LAC performance for 3 query types: FR-HOT, SPSP, RW.

Table 4 shows that ordering is a dominant factor in the performance. However, LAC nicely exploits the underlying locality improvements and outperforms GCLOCK. The new graph ordering

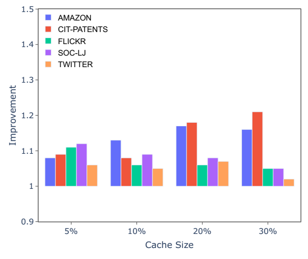
[12]LAC's page fault improvements highly correlate with latency improvements.



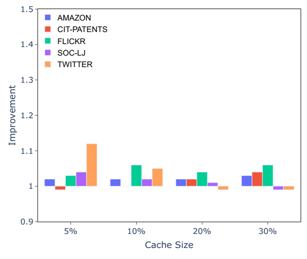
(a) FR-ALL



(b) FR-HOT



(c) SPSP



(d) RW

Figure 9: LAC's latency improvement over GCLOCK in Neo4j.

approach is shown to outperform Gorder in [7]. While comparing the improvements in Figure 7 and Table 4, our results show that improved locality increases LAC's performance.

5.4 LAC Performance in native Neo4j implementation

In this section, we present the effectiveness of LAC in Neo4j. We again focus on latency comparisons. LAC's performance vis a vis native Neo4j cache replacement policy (which is GCLOCK) for all graph and query types is given in Figure 9.

The results show that LAC's performance improvements in Neo4j track those of the simulation experiments. LAC outperforms native Neo4j by up to 1.23× for almost all cases. Although the trends are similar to the simulation results, the improvements are lower in the Neo4j implementation. These can be explained as follows:

① As noted in Section 4.2, Neo4j shares the page cache with all store types including auxiliary data while in our simulated setup, the cache space is occupied by only vertex and edge pages. Consequently, the cache utilization for vertex and edge page accesses is lower in Neo4j than in the simulation.

② Our simulated setup follows GDBMS storage design as described in Section 4.1. Neo4j has a unique storage design where each edge page holds approximately one-fourth the number of edges of the simulation. Therefore, the number of disk pages in Neo4j is four times greater for the same dataset. When there are more edge pages, the cache contains, at any given time, more consecutive edge pages. Fetching more consecutive edge pages into the cache results in more pages whose *DCF* values become 1. As we explain in Section 5.3.1, this situation results in reducing the impact of *DCF* in LAC's calculations because these pages' increment and decrement values (SET_{p_i}) become the same and LAC and GCLOCK behaviours start to converge.

1-hop query types are the most affected by this situation. In these query types, the number of edge traversals is at most equal to the degree of the start vertex. Since Neo4j keeps far more edge pages in the cache and shares it with auxiliary data pages, the number of vertex pages it can cache is lower. This negatively impacts neighbourhood detection in LAC. The RW query type is a good example to explain the situation – it is the least affected by this organization. That is because it requires traversing to the next hop at each step and this increases the number of **distinct** page accesses. Consequently, page density becomes less significant.

Due to the edge page capacity of Neo4j, the edge store contains hundreds of thousands of edge pages for large graph. Since the cache size is set as a percentage of the data size, even with small cache percentages, cache is large and numerous consecutive edge pages are fetched into it, resulting in their *DCF* values becoming 1. As explained earlier, this scenario causes LAC to perform similarly to GCLOCK when promoting pages. For demotions, however, this situation creates a worst-case scenario for LAC. Because LAC demotes pages with *DCF* values of 1 as minimally as possible, when there are many such pages, it requires more than one sweep of the circular list to locate an eviction candidate with an *RC* value below 1. This adds runtime overhead to the policy. Although LAC still outperforms in terms of reducing the number of page faults, its latency increases and its relative advantage is slightly lower.

6 CONCLUSIONS

In this paper, we show that data reuse in graph applications is complex and the well-known and widely used cache replacement policy, GCLOCK, is insufficient in many cases.

Since topology-aware ordering of graph data enables intelligent caching approaches that would further improve the cache hit rates, we use the data layouts obtained by a graph ordering scheme and propose LAC as a graph locality-aware cache replacement policy. LAC utilizes the cache by maintaining the residency of pages in it that are likely to be accessed and reused together based on the structural properties of input graphs. We show that the spatial locality that is captured in disk pages offers temporal locality for the subsequent accesses of cache pages, and this information can be used in better replacement decisions.

We evaluate LAC against its competitors for the input graphs with different structural properties while running various query types on the underlying simulated system where different cache sizes are set. We show that LAC outperforms its competitor in most cases both in the simulated setup and in the real system. Currently, LAC is applicable to read-only workloads. We have future work plans to use LAC's methodology in prefetching where updates can also be handled and to enhance the implementation of LAC within Neo4j. This includes exploring the integration of background eviction mechanisms to further optimize cache management.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grants RGPIN-2024-03993 and RGPIN-2024-04657. We thank Lori Paniak for supporting the computing infrastructure used for this work.

REFERENCES

[1] [n. d.]. Graph Data Platform | Graph Database Management System | Neo4j. https://neo4j.com/

[2] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Ozgur Ulusoy. 2015. Graph Aware Caching Policy for Distributed Graph Stores. In *IEEE Int. Conf on Cloud Engineering*. 6–15.

[3] Reza Bakhshandeh, Mehdi Samadi, Zohreh Azimifar, and Jonathan Schaeffer. 2011. Degrees of Separation in Social Networks. *Proceedings of the 4th Annual Symposium on Combinatorial Search*.

[4] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. https://doi.org/10.1147/sj.52.0078

[5] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline operator. *Proceedings 17th International Conference on Data Engineering* (2001), 421–430. https://api.semanticscholar.org/CorpusID:5812098

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proc. USENIX 2013 Annual Technical Conf.* 49–60.

[7] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proc. VLDB Endowment* 17, 6 (2024), 1405–1417. https://doi.org/10.14778/3648160.3648179

[8] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 619–630. https://doi.org/10.1145/2723372.2742786

[9] Priyank Faldu, Jeff DIamond, and Boris Grot. 2020. A closer look at lightweight graph reordering. *Proc. Int. Symp. on on Workload Characterization* (2020), 1–13. https://doi.org/10.48550/arxiv.2001.08448

[10] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-specialized cache management for graph analytics. *Proc. IEEE Int. Symp. on High-Performance Comp. Architecture* (2020), 234–248.

[11] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2022. Kùzu Database Management System Source Code. https://github.com/kuzudb/kuzu

[12] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu Graph Database Management System. In *Proc. 13th Biennial Conf. on Innovative Data Systems Research*.

[13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proc. 19th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. 77–85. https://doi.org/10.1145/2487575.2487581

[14] Nicolaus Henke, Jacques Bughin, Michael Chui, J. Manyika, Tamim Saleh, and Bill Wiseman. 2016. The Age of Analytics: Competing in a data-driven world. https://api.semanticscholar.org/CorpusID:196173558

[15] Imranul Hoque and Indranil Gupta. 2012. Disk Layout Techniques for Online Social Network Data. *IEEE Internet Comput.* 16, 3 (2012), 24–36.

[16] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. 37th Annual Symp. on Computer Architecture*. 60–71. https://doi.org/10.1145/1815961.1815971

[17] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Comput.* 20 (1998), 359–392.

[18] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*. 31–46.

[19] Victor F. Nicola, Asit Dan, and Daniel M. Dias. 1992. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *Proc. 1992 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems* 20, 1 (1992), 35–46. https://doi.org/10.1145/149439.133084

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symp. on Networked Systems Design & Implementation*. 385–398.

[21] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Rec.* (1993).

[22] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proc. ACM Manag. Data* 2, 3, Article 125 (2024). https://doi.org/10.1145/3654928

[23] Jordi Petit. 2004. Experiments on the minimum linear arrangement problem. *ACM J. Exp. Algorithmics* 8, Article 2.3 (2004).

[24] Ira Pohl. 1969. Bi-directional and heuristic search in path problems. https://api.semanticscholar.org/CorpusID:61081057

[25] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endowment* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[26] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2–3 (2019), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[27] Alan Jay Smith. 1978. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.* 3 (1978), 223–247.

[28] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Vicente Calisto, Benjamín Farías, Sebastián Ferrada, Tristan Heuer, Aidan Hogan, Gonzalo Navarro, Alexander Pinto, Juan Reutter, Henry Rosales, and Etienne Toussiant. 2024. MillenniumDB: A Multi-modal, Multi-model Graph Database. In *Companion of ACM SIGMOD Int. Conf. on Management of Data*. 496–499. https://doi.org/10.1145/3626246.3654757

[29] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1813–1828. https://doi.org/10.1145/2882903.2915220

[30] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. 2013. Fast iterative graph computation with block updates. *Proc. VLDB Endowment* 6, 14 (2013), 2014–2025.

[31] Abdurrahman Yaşar, Buğra Gedik, and Hakan Ferhatosmanoğlu. 2017. Distributed block formation and layout for disk-based management of large-scale graphs. *Distrib. Parall. Databases* 35, 1 (2017), 23–53. https://doi.org/10.1007/s10619-017-7191-3

[32] Ji-Tae Yun, Su-Kyung Yoon, Jeong-Geun Kim, and Shin-Dug Kim. 2020. Access pattern-based high-performance main memory system for graph processing on single machines. *Future Generation Comput. Syst.* 108 (2020), 560–573. https://doi.org/10.1016/j.future.2020.03.015

[33] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2017. Caching at the web scale. *Proc. VLDB Endowment* 10, 12 (2017), 2002–2005.

[34] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *Proc. 2017 IEEE Int. Conf. on Big Data*. 293–302. https://doi.org/10.1109/BigData.2017.8257937

[35] Da Zheng, Randal Burns, and Alexander S. Szalay. 2013. Toward millions of file system IOPS on low-cost, commodity hardware. In *Proc. 2013 ACM/IEEE Conf. on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503225

[36] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proc. 13th USENIX Conf. on File and Storage Technologies*. USENIX Association, 45–58.

[37] Ningnan Zhou, Xuan Zhou, Xiao Zhang, Shan Wang, and Ling Liu. 2016. An I/O-Efficient Buffer Batch Replacement Policy for Update-Intensive Graph Databases. In *Proc. 21st Int. Conf. on Database Systems for Advanced Applications*. Springer-Verlag, 234–248. https://doi.org/10.1007/978-3-319-32049-6_15

.