Article

In-Memory Caching for Enhancing Subgraph Accessibility

Kyoungsoo Bok ¹, Seunghun Yoo ², Dojin Choi ², Jongtae Lim ² and Jaesoo Yoo ^{2,*}

¹ Department of SW Convergence Technology, Wonkwang University, Iksandae 460, Iksan, Jeonbuk 54538, Korea; ksbok@wku.ac.kr

² Department of Information and Communication Engineering, Chungbuk National University, Chungdae-ro 1, Seowon-Gu, Cheongju, Chungbuk 28644, Korea; yshun2487@chungbuk.ac.kr (S.Y.); mycdj91@chungbuk.ac.kr (D.C.); jtlim@chungbuk.ac.kr (J.L.)

* Correspondence: yjs@chungbuk.ac.kr; Tel.: +82-43-261-3230

Received: 9 June 2020; Accepted: 7 August 2020; Published: 9 August 2020

Abstract: Graphs have been utilized in various fields because of the development of social media and mobile devices. Various studies have also been conducted on caching techniques to reduce input and output costs when processing a large amount of graph data. In this paper, we propose a two-level caching scheme that considers the past usage pattern of subgraphs and graph connectivity, which are features of graph topology. The proposed caching is divided into a used cache and a prefetched cache to manage previously used subgraphs and subgraphs that will be used in the future. When the memory is full, a strategy that replaces a subgraph inside the memory with a new subgraph is needed. Subgraphs in the used cache are managed by a time-to-live (TTL) value, and subgraphs with a low TTL value are targeted for replacement. Subgraphs in the prefetched cache are managed by the queue structure. Thus, first-in subgraphs are targeted for replacement as a priority. When a cache hit occurs in the prefetched cache, the subgraphs are migrated and managed in the used cache. As a result of the performance evaluation, the proposed scheme takes into account subgraph usage patterns and graph connectivity, thus improving cache hit rates and data access speeds compared to conventional techniques. The proposed scheme can quickly process and analyze large graph queries in a computing environment with small memory. The proposed scheme can be used to speed up in-memory-based processing in applications where relationships between objects are complex, such as the Internet of Things and social networks.

Keywords: graph caching; subgraph; prefetching; replacement; used cache; prefetched cache

1. Introduction

Users have created and utilized a large amount of graph data due to the advancement of internet technology and mobile devices [1,2]. Graph data have been utilized to represent interactions and relationships between objects through vertices and edges in various fields, such as social network service (SNS), semantic web, bio-information systems, and the Internet of Things (IoT) [3–6]. These graph data are used in pattern mining, prediction, and recommendations through relationships between objects as well as the information of all vertices [7–11]. In social networks, graph data are used to determine user influence through human network analysis. In bio-information systems, graph data are utilized for disease diagnosis and new drug developments. E-commerce, such as online shopping and movie and music content services, employ graph data to recommend items to consumers.

To process and analyze a large amount of graph data, various studies on distributed graph processing [12–15], graph pattern matching [16–18], and incremental graph processing [19–22] have been conducted. Recently, various studies on in-memory caching have been conducted to improve

system performance effectively [23–27]. In-memory caching is a technique to reduce data access costs by loading some of the original data present on disk into memory. In-memory caching typically uses pre-fetching techniques to store data in memory that is expected to be used in the future since not all data can be loaded into memory. We use a replacement strategy to change cached data if there is not enough caching space allocated to memory. The graph expresses a complex relationship (multiple relationships) between objects through vertices and edges. Therefore, in applications such as SNS and the IoT, graph processing tends to approach associated subgraphs rather than accessing a single vertex or edge. In addition, when the size of the graph is large or the connection relationship of the graph is complicated, a situation occurs in which all of the subgraphs an individual wishes to cache are not loaded into memory. In order to reduce the cost of access to subgraphs, in-memory caching schemes are required, taking into account the characteristics of the use of the graphs. To do this, it is necessary to identify frequently used subgraph patterns and to load highly available subgraphs into memory. Caching neighboring subgraphs is necessary to reduce disk access costs because access to specific vertices in the process of query processing and analysis requires access to the subgraph connected to the edge. For the in-memory caching of graph data, the graph connectivity should be taken into consideration. A subgraph that is likely to be accessed spatially is predicted for future subgraph access considering connectivity. When a certain vertex is accessed, the neighbor vertices are likely to be accessed next through the edges connected to the vertex.

The existing graph caching schemes have proposed a method that stored all neighbor vertices in the cache when accessing an arbitrary vertex [28,29]. In [28], general Least Recently Used (LRU) was used as a graph replacement strategy, but the same processing method used for general objects was employed without consideration of the graph data topology. In [29], a replacement policy was proposed according to the topology characteristics of graph data. It cached the neighbor vertices of the accessed vertex since the neighbor vertices were likely to be used again in the future. It also proposed an algorithm that replaced a subgraph with a low weight with a new subgraph when the memory was full by assigning a weight during graph caching. However, subgraphs that were unlikely to be used later occupied the memory space by making all connected vertices memory resident when caching graph data. Thus, memory overload was likely to occur because all neighbor vertices were memory resident when accessing a vertex with many neighbors. Furthermore, since the existing studies replaced a large amount of graphs at once, it even cached subgraphs that were unlikely to be accessed. The existing studies did not consider neighbor vertices employed together when subgraphs were used during graph data processing. In general, frequently used subgraphs are likely to be used again. Thus, a high-usage subgraph should have a higher survival rate in the memory than that of a low-usage subgraph. Moreover, when graph data were cached in the memory, the neighbor vertices were all cached. For example, for a famous person on Facebook who has tens of millions of friends, the neighbor friend information must be stored in memory. If access to the vertex with many neighbor neighbors occurs frequently, memory can be overloaded or many input/outputs (I/Os) may occur in the memory.

In this paper, we propose a two-level caching strategy that stores subgraphs that are likely to be accessed in consideration of the usage pattern of subgraphs to prevent the caching of low-usage subgraphs and frequent subgraph replacement in the in-memory. A cache level is divided into two levels: a used cache that stores a subgraph by assigning a weight according to the frequently used subgraph pattern and a prefetched cache that stores the neighbor subgraph of a recently used subgraph. Subgraph usage patterns are extracted by means of the query management table and Frequent Pattern-tree (FP-tree) to extract and manage subgraph usage patterns by utilizing the usage history information of subgraphs. The neighbors connected to edges with a high history information value among neighbor vertices during the current query are loaded in the prefetched cache utilizing the history information of the edges. We also propose a strategy that manages the cached subgraph in each cache and replaces the cached subgraph that is unlikely to be used with a new subgraph when the memory is full. The subgraph of the used cache is managed through a time-to-live (TTL) value

that is assigned when the subgraph is assigned to the memory. It uses a strategy that replaces the subgraph with a low TTL value with the newly loaded subgraph. The subgraph in the prefetched cache is managed by the “first-in, first-out” (FIFO) strategy. When the memory is full, the first-in subgraph is replaced with a newly cached subgraph. Once a hit occurs at the prefetched cache, the subgraph is migrated to the used cache and managed after assigning the TTL value.

This paper is organized as follows. Section 2 describes the existing in-memory graph data cache management schemes. Section 3 introduces the in-memory caching and management scheme using subgraph patterns and graph usage history information. Section 4 verifies the performance of the proposed scheme through experiments. Finally, Section 5 presents the conclusion and future study direction of this paper.

2. Related Work

The in-memory caching scheme has been utilized to process graphs fast in the distributed memory environment. Ran et al. proposed a distributed in-memory graph storage system called NYNN to analyze large-scale graphs created in social networks [28]. This system took large-scale graphs, frequent access, and updates into consideration, which are the characteristics of a social network graph. They proposed a method that loaded the neighbors of an accessed subgraph into the memory for the first subgraph access to the subgraph used in arbitrary vertex access. The overall graphs are divided into subgraphs, which are then divided and stored as a form of data server. The information about the subgraphs and data server are stored and managed by the subgraph machine mapping table (SMMT) of the name server. It can reduce data I/O costs by prefetching subgraphs in advance. The first access to graph data is difficult to predict due to the access to the arbitrary vertex. However, the next access subgraphs are likely to be neighbor vertices on the basis of the previous requested vertex. Thus, the adjacent vertices of the requested vertex are all prefetched to the memory.

Aksu et al. proposed a graph data caching policy for distributed graph processing on a big data platform [29]. This policy took metadata size, the number of hops between neighbors, and communication according to distributed processing into consideration by analyzing the graph data topology. It proposed a cache policy that cached neighbor vertices and vertices with multiple hops by estimating the neighbor vertices of the vertex currently accessed for the future access to graph data as a prefetching strategy. They also proposed clock-based graph-aware (CBGA) caching as a caching replacement strategy. The graph data caching policy proposed by [29] also caches neighbor vertices in the memory, the same as in [28]. In addition, not only single-hop neighbors but also multi-hop neighbor subgraphs are loaded to the memory. Whenever subgraphs are loaded to the memory, a TTL value is assigned to them. The TTL value is determined by the communication cost, the number of hops, and size. The TTL value decreases by one as time passes. Subgraphs whose TTL value becomes zero are removed from the memory. If there are no subgraphs whose TTL value is zero, the subgraphs with the lowest TTL value are removed from the memory first and new subgraphs are loaded.

Wang et al. proposed a semantic cache called GraphCache (GC) to reduce sub-iso testing in filter-then-verify (FTV) [23]. In GC, the cache manager consists of the cache stores and the window stores for subgraph and supergraph queries. The cache stores manage a component that stores copies of cached queries with their result sets, a combined index, and component storing statistics for each cached query. The combined index has a single index to expedite subgraph/supergraph matching of future queries against past queries and then provides lower disk space and I/O overhead. The window stores manage a component that stores new graph queries and their result sets and component storing statistics for each query in the previous component. GC was used to test various cache replacement strategies such as POP, PIN, and PINC, as well as LRU, where POP is popularity-based ranking, PIN is POP + number of sub-iso tests, and PINC is PIN + sub-iso test costs.

Cicotti et al. proposed a new caching scheme to reduce the computation and communication in a parallel breadth first search (BFS) [30]. This scheme caches the remote vertices previously visited and checks the cache to reduce communication. In the parallel BFS, many messages are sent to notify a

process of different candidate parents for the same vertex. If the remote vertex is stored in the cache, it does not send a message, otherwise the cache is updated, and the message is sent. Each process maintains a vertex cache in memory. In the remote caching, each server process maintains the cache. Several servers are connected to the same switch to share a cache server.

Jiang et al. proposed ID caching technology (IDCT) for graph databases for accelerating the queries on graph database data and avoiding redundant graph database query operations [31]. Each cache stores cache query, ID array, the data created and last used, and the times of access. Because not all caches are accessed frequently, IDCT manages landing status for a month or months without the same query. IDCT change queries over time to get the best cache for the occasional query. If the cache data occurs less than three times within 24 h, caching is not required. If the number of caches is too high or a memory overflow occurs, the inefficient caches are deleted from the array according to the least common used (LFU) to replace unused cache in recent years.

Braun et al. proposed data stream matrix (DSMatrix) to extract a frequent subgraph pattern [32]. DSMatrix is a 2D structure that represents whether an edge in a graph is generated by 1 or 0 in a small memory space. DSMatrix uses two strategies to detect a frequent pattern: recursive-FP-tree and FP-trees for only frequent singletons. Graph are stored in the DSMatrix and then an FP-tree is constructed for all edges by using the recursive-FP-tree scheme. All frequent patterns are detectable by recursively constructing the FP-tree. In the FP-trees for only frequent singletons, an FP-tree is first constructed with regard to all edges, and then each node is visited using the depth-first search scheme to calculate the number of occurrences of frequent patterns. This scheme can detect frequent patterns at a lower cost than that of the recursive-FP-tree since FP-trees are constructed for only a singleton edge.

In [28,29], the graph topology is considered to perform the graph caching. The study in [28] considers large-scale graphs as well as frequent access and updates and prefetches the neighbor vertices when accessing certain vertexes. The study of [29] prefetches the neighbor vertices connected to a single hop or multi-hops for a specific vertex and assigns the TTL value through considering size and the number of hops to cached subgraph. However, if there are many adjacent neighbor subgraphs, this may cause frequent graph replacement inside the memory or memory overload. When there are many adjacent neighbor data, frequent data replacement inside the memory and memory overload occur. The authors of [23,30,31] proposed schemes to improve query processing performance by caching the existing query results. However, when processing a new query or a query for random access, it is expensive to access a subgraph that is not stored in a cache or a subgraph connected with query results. In the existing scheme, if there are many adjacent neighbor subgraphs, frequent data replacement may occur inside the memory or there may be memory overload. Moreover, side effects may occur in which the subgraph that will be used in the future will be replaced or the subgraph that will not be used will continue to reside in the memory. To solve this problem, this paper uses the FP-tree [32] to extract a frequent subgraph pattern. In addition, a caching algorithm is proposed in which information about the usage pattern and access history is applied to the caching system. Due to the use of the LRU mode designed for general data, subgraphs that will be used in the memory may be replaced. Moreover, when there are many adjacent neighbor subgraphs, frequent graph replacement inside the memory and memory overload occur.

3. The Proposed Graph Caching Scheme

3.1. Overall Structure

Graph data are processed rapidly by prefetching subgraphs on a disk to the memory in advance. The existing studies did not consider neighbor vertices employed together when subgraphs were used during graph data processing. Moreover, when subgraphs were cached in the memory, the neighbor vertices were all cached. Thus, this paper proposes a two-level caching scheme that considers the usage pattern and access frequency of subgraphs. The proposed scheme is configured with a two-level

caching structure managed by different caches by caching the frequently used subgraphs and neighbor vertices of the accessed subgraph.

Figure 1 shows the proposed overall structure. The structure of the proposed in-memory caching scheme is divided into a cache manager and a two-level cache. The cache manager stores query histories used in the query management table and generates frequent subgraph patterns through an FP-tree. In addition, it generates a subgraph access pattern and assigns a TTL value for managing cached graphs. The two-level cache is divided into a used cache (UC) and a prefetched cache (PC). In the used cache, subgraphs that are likely to be used among previously used subgraphs are cached. In the PC, unused subgraphs that are likely to be used in the next query are cached. If subgraphs stored in the cache are actually used, they are migrated to the used cache.

Figure 1. Overall structure.

Graph $G = (V, E)$ is represented as a set of vertices V and a set of edges E . Vertex V consists of (VID, VP) , where VID is the vertex identifier and VP represents the type or label, etc. as the attribute information of the vertex. Edge E consists of (EID, EP) , where EID is the edge identifier, and EP value represents the edge weight and type, etc. The graph is implemented as an adjacency list consisting of a vertex table and an edge list [33,34]. The vertex table manages information about all vertices and keeps pointers to VID s, attributes, and edge lists connected to vertices for each element. The edge list is implemented as a link list and maintains the vertices connected to the EID , attributes, and vertices for each element. Since UC and PC are given TTL values, they are implemented as ordered linked lists according to TTL values. Since PC is managed by FIFO structure, they are implemented as queues implemented by linked lists.

When a graph request is received, the system inspects whether the requested subgraphs are cached in the UC or the PC. If subgraphs are found, subgraphs inside the memory are used. Otherwise, subgraphs are loaded from the disk. Not only replying to the query, but also storing the requested query in the query history management table is performed at the same time. The FP-tree is created through the query history management table, and the TTL value is calculated through the query pattern, thereby assigning the TTL value to the subgraph stored in the used cache. Algorithm 1 shows

the subgraph access process. When there is a request for a subgraph, we first determine whether the requested subgraph *rsubGraph* exists in memory. If *rsubGraph* exists in memory, the requested subgraph is passed on to *resultGraph* via *readSubgraph()*. Otherwise, we determine whether a cached subgraph exists in UC or PC. If *rsubGraph* exists in UC or PC, we forward the requested subgraph to *resultGraph*. If *rsubGraph* exists in UC, we increase the TTL value to keep the corresponding subgraph constant, otherwise we decrease the TTL value. If *rsubGraph* does not exist in UC and PC, we load the subgraph into memory by performing *loadSubgraph()*.

Algorithm 1 *subgraphAccess(rsubGraph, UC, PC)*

Input

rsubGraph: request subgraph

UC: used cache

PC: prefetched cache

Output

resultGraph: result subgraph

if *existMem(rsubGraph)*

resultGraph $\leftarrow$ *readSubgraph(rsubGraph)*

else

if *hitUC(rsubGraph)*

resultGraph $\leftarrow$ *readSubgraph(rsubGraph)*

increaseTTL(*resultGraph*)

else if *hitPC(rsubGraph)*

resultGraph $\leftarrow$ *readSubgraph(rsubGraph)*

decreaseTTL(*resultGraph*)

else

resultGraph $\leftarrow$ *loadSubgraph(rsubGraph)*

decreaseTTL(*resultGraph*)

3.2. Graph Usage Pattern

Frequently used subgraphs are likely to be used again. In the case of social network services, users tend to frequently search for information that has recently become an issue, or they frequently access the latest documents. In addition, video services such as YouTube tend to frequently access or search for content that users watch a lot or that are produced by celebrities. In other words, most applications that use graphs are likely to use recently frequently used subgraphs again. Therefore, it is necessary to extract the usage patterns of graphs and load frequently used subgraphs into memory in order to improve access performance to subgraphs. In this paper, a weight was assigned to each subgraph on the basis of the extracted pattern. A high weight is assigned to vertices that were used simultaneously to reside in the memory for a long period of time. Whenever a subgraph query is received, corresponding query information is stored in the query history management table. The query management table is managed by a queue structure, and queues are deleted sequentially according to FIFO strategy if the table is full. The FP-tree is created according to the query history management table. A list of frequently accessed vertices is found while searching the table, and is sorted in descending order. According to the sorted list of frequency vertices, queries are searched through the second search and the FP-tree is created through a recursive call. The FP-tree is constructed through the query history in the query history management table, and the pattern weight (PW) is extracted.

If the subgraph queries are requested as shown in Figure 2, the query histories are managed and then an FP-tree is generated, as shown in Figure 3. Figure 3a shows a query history management table to store information about the edges used in past queries. Here, the QID is the query identifier and edge information represents the information of the edges used in the query. When a query is received, edge information that correspond to each query is stored. For example, if Q_1 is requested, edges e_1, e_2, e_3, e_4 , and e_5 are stored in the query history management table. We generated FP-trees as shown in

Figure 3b using the information of the edges stored in Figure 3a. FP-trees were created to extract the subgraphs used by past queries. Since e_1 and e_3 occurred most frequently, the FP-tree was created on the basis of e_1 and e_3 . Using the generated FP-tree, we detected the frequent subgraphs by performing the depth first search (DFS) proposed in [32].

Figure 2. Cont.

Figure 2. Subgraph query: (a) Q_1 ; (b) Q_2 ; (c) Q_3 ; (d) Q_4 ; (e) Q_5 .

Figure 3. Query history and FP-tree: (a) query history management table; (b) FP-tree.

3.3. Subgraph Caching

We present a cache structure that divides a cache level into a UC and a PC to efficiently access subgraphs that were used frequently and subgraphs that are likely to be used in the future, which are managed differently. Since the criteria of loading subgraphs to the memory differ for the two levels, we stored subgraphs in different caches according to the criteria to manage subgraphs easily. When a graph query is requested, previously used subgraphs are likely to be used again. Thus, the UC caches the used subgraphs and stores subgraphs used frequently in the memory for a long period of time. If the cache space with a limited size is full, the cached subgraph must be replaced. When the UC space is full, the UC replaces the subgraph according to the TTL value. If the number of vertices included in the subgraph is high, it is likely that the vertices used in other queries will be included. In addition, the subgraphs frequently used in past queries are likely to be used in the future. Therefore, the TTL value for a subgraph depends on the number of vertices contained in the subgraph and the frequency of use of the subgraph. Here, NV_i is the number of vertices included in the subgraph and PW_i is the pattern weight given in the process of detecting frequent patterns through the FP-tree as the usage pattern on the subgraph. Subgraphs with large TTL values are less likely to be replaced. Cached subgraphs may differ in the importance of the number of vertices and frequency of use, depending on application. Equation (1) uses α to assign weights to the number of vertices and pattern weight. Here, α ranges from 1 to 1.

$$TTL_i = \alpha NV_i + (1 - \alpha)PW_i \quad (1)$$

Figure 4 shows the subgraph patterns used by three queries: Q_1 , Q_2 , and Q_3 . If we detect a frequent pattern by constructing an FP-tree for the subgraph used by each query, we can see that G_1 with vertices v_1 , v_2 , and v_3 is frequently generated by the frequent pattern. When queries Q_1 , Q_2 , and Q_3 are received, the vertices in each query are cached in the UC. Table 1 presents the TTL values, where VID is a vertex identifier and TTL is a TTL value assigned to each vertex. Each subgraph calculates a TTL value using Equation (1) and assigns the calculated TTL value to each information

contained in the subgraph. The TTL values of vertices v_1 , v_2 , and v_3 are high because they belong to G_1 , which has a high access pattern.

Figure 4. Graph query pattern.

Table 1. Time-to-live (TTL) values of vertices.

VID	TTL
v_1	30
v_2	30
v_3	30
v_4	25
v_5	25
v_6	22
v_7	22
v_8	22
v_9	22
v_{10}	22
v_{11}	0
v_{12}	0
v_{13}	0

Query processing and analysis of graphs perform operations with neighboring vertices connected by edges incident to a particular vertex. In other words, since the graph computation is done using the relationship between vertices, it is not performed using only one vertex or edge, but also uses the other vertices associated with a particular vertex. Therefore, when we approach a particular vertex, we approach the vertices adjacent to it. The neighbor vertices of the requested subgraph are cached in the PC. The number of subgraph accesses that were used together is stored in each graph edge. For example, if vertex v_1 is used along with vertex v_2 , the number of accesses of the edge that connects vertices v_1 and v_2 is increased by 1. If the graph is complex and there are too many vertices adjacent to a particular vertex, all neighbors of the particular vertex cannot be stored in a cache space with a limited size. Therefore, when approaching a particular vertex, it selectively caches neighboring vertices in consideration of accessibility. When a query is requested, the neighbor vertices of the used subgraph that have a high frequency access pattern are also cached along with the used memory in the memory. Top-k vertices that have a high access pattern are singled out using the number of accesses stored in

the graph edges and cached in the memory. Furthermore, single-hop and multi-hop neighbor vertices are cached in the memory. For the multi-hop neighbor vertex cache, vertices that have a high access history of vertices selected from the previous hop are chosen.

When a query is requested, as shown in Figure 5, vertices that have a high frequent access pattern are searched among neighbor vertices. When k is 2, vertices v_7 and v_{13} are selected and cached in the memory. On the other hand, v_1 , v_4 , and v_8 , which are other neighbor vertices, are not cached. When multi-hop is considered, vertex v_{12} , which has a high access history with vertex v_7 , and vertex v_{14} , which has a high access pattern with vertex v_1 , are selected and cached together.

Figure 5. Neighbor vertex caching according to graph query.

3.4. Cache Replacement

When the cache space is full, no more subgraphs are cached. Thus, a strategy that removes a subgraph from the memory or replaces a subgraph with a new subgraph is needed. The UC caches subgraphs that were used before. When a query request is received, the subgraphs are cached in the UC in case a cache miss or hit occurs in the PC. When the UC space is full, the UC replacement strategy is performed. Algorithm 2 shows the replacement algorithm of UC. We assume a new subgraph $newSubgraph$ exists that we want to cache in UC when UC is full. Since UC is controlled by TTL values, we perform $readTTL()$ to check the TTL values of each subgraph i cached in UC and $newSubgraph$. If the TTL of the $newSubgraph$ is greater than the TTLs of the subgraphs cached in the UC, we replace the subgraph $minSubgraph$ with the lowest TTL value with the $newSubgraph$.

Algorithm 2 replaceUC ($newSubgraph$, $ucSubgraph$)

Input

$newSubgraph$: new subgraph

$ucSubgraph$: subgraph cached in UC

Output

$newucSubgraph$: subgraph changed by replacement in UC

while($ucGraph$)

$currentTTL \leftarrow readTTL(ucSubgraph, i)$

$newTTL \leftarrow readTTL(newSubgraph)$

 if $currentTTL < newTTL$

$tempList \leftarrow addtempList(i)$

if $existtempList()$

$minSubgraph \leftarrow findMinTTL(tempList)$

$newucSubgraph \leftarrow replaceSubgraph(tempList, minSubgraph)$

Table 2 shows the UC replacement strategy. When vertices v_1 , v_3 , and v_6 are requested as Figure 6, vertices v_1 and v_3 are hit so that the TTL values are updated and TTL values of other vertices are decreased. Furthermore, vertex v_6 has a cache miss and is replaced with vertex v_5 , which has the lowest TTL value among the subgraphs in the cache. Thus, vertex v_5 is deleted and vertex v_6 is stored.

Table 2. Example of UC replacement.

VID	v_1	v_2	v_3	v_4	v_5	v_6
Initial TTL	10	3.5	15	21	5	-
Updated TTL	27	34	30	20	4	-
Final TTL	27	34	30	20	-	17

Figure 6. New requested query in UC.

When a query request is received, since the neighbor vertices of the used vertex are likely to be used next time, the PC caches the neighbor vertices of the used subgraph. The PC manages the cached subgraph according to the FIFO method. Algorithm 3 shows the PC replacement algorithm. The cache replacement in PC occurs when a cache hit occurs in the PC and the subgraph cached in the PC moves to the UC, and when a new graph newSubgraph is requested and requires caching neighboring subgraphs. If the cache hit is a cached subgraph in the PC, resultGraph is generated. Therefore, to replace the cached subgraph in the PC, the resultGraph is determined to exist. If resultGraph exists, we first move the hit subgraph from the PC to the UC by performing moveUC() and perform the readSubgraph() to store the neighboring subgraphs connected to the newSubgraph in the PC. If there is no cache hit in the PC but there is a request for newSubgraph, we delete the existing subgraph by performing deleteSubgraph() and store the new subgraph by performing addSubgraph().

Algorithm 3 replacePC(newSubgraph, resultGraph, pcSubgraph)

Input

newSubgraph: new subgraph
 resultGraph: results hit by rsubGraph on PC
 pcSubgraph: subgraph cached in PC

Output

newpcSubgraph: subgraph changed by replacement in PC

if existresultGraph(resultGraph)

moveUC(resultGraph)

tempSubgraph $\leftarrow$ readSubgraph(newSubgraph)

newpcSubgraph $\leftarrow$ newpcSubgraphaddPC(tempSubgraph)

else

newpcSubgraph $\leftarrow$ deleteSubgraph(pcSubgraph)

tempSubgraph $\leftarrow$ readSubgraph(newSubgraph)

newpcSubgraph $\leftarrow$ addSubgraph(tempSubgraph)

Figure 7 shows the PC replacement strategy. We suppose that a new query is requested as shown in Figure 8 when a vertex v_1 is stored in the PC and vertices v_2 and v_3 are stored in the UC. Because vertices v_2 and v_3 are hit, as shown in Figure 7a, these vertices are deleted from the PC and are migrated to the UC. If the PC spaces are available due to the movement of v_2 and v_3 , vertices v_4 and v_7 with the high frequent access pattern are prefetched to PC, as shown in Figure 7b. If the PC is full, the vertices are replaced by the FIFO, as shown in Figure 7c.

Figure 7. Prefetched cache replacement: (a) initial states; (b) cache hit; (c) cache replacement.

Figure 8. New requested query in the PC.

4. Performance Evaluation

The superior performance of the proposed scheme was proven through a performance evaluation comparison between the proposed and existing schemes. In this performance evaluation, we compared NYNN [28] and CBGA [29]. The performance evaluation environment is presented in Table 3. Simulations were conducted through the Java program on a personal computer running an Intel Core™ i5-4460 CPU @ 3.20 GHz, RAM 8.00 GB, Microsoft Windows 10 64-bit operating system. For the experimental data, we used three datasets from Twitter [35], Weki-Talk [36], and YouTube [37], as presented in Table 4. The Twitter dataset was provided by Arizona State University, and the Weki-Talk and YouTube datasets were provided by Stanford University. The number of subgraphs requested ranged between 1000 and 10,000.

Table 3. Performance evaluation environment.

Feature	Description
Processor	Intel Core i5-4460 CPU @ 3.20 GHz
Memory	8 GB
Disk	1 TB
Program language	Java

Table 4. Dataset.

Dataset	No. of Vertices	No. of Edges
Twitter	11,316,811	85,331,846
Weki-Talk	2,394,385	5,021,410
YouTube	1,134,890	2,987,624

Threshold α should be determined to decide TTL value assigned from the UC proposed in this paper. The α value that produced a high hit ratio was searched by changing the α value that was multiplied by the number of vertices included in the pattern as well as the pattern weight in Equation (1). Performance evaluation was conducted by changing the α value from 0.1 to 0.9 by 0.05 increments, when 5000 queries were generated. Figure 9 shows the evaluation results of hit ratio by changing α value. As shown in Figure 9, the highest hit ratio was revealed when the α value was 0.30. Thus, $\alpha = 0.3$ was used in performance evaluation because it showed the best hit ratio.

Figure 9. Hit ratio according to α value.

Performance evaluation on hit ratio and processing time was conducted under three scenarios: when only the UC proposed in this paper was used, when only the PC was used, and when two-level cache was used. Here, 5000 queries were conducted with regard to Twitter, Weki-Talk, and YouTube datasets. For TTL weight, we used $\alpha = 0.3$ in the performance evaluation. Figure 10 shows hit ratio when each of the caches were used. The highest hit ratio was exhibited when, out of the three scenarios, two-level cache was used. When only the UC was used, frequently used subgraph patterns were cached in the memory, but overall performance was low because it did not take future access data into consideration. In contrast, when the PC was used, overall performance was also low because it did not take frequently used patterns into consideration, but instead only future access data were cached. The performance evaluation results showed that the hit ratio of the two-level cache was improved by 143% over the scheme using only UC, and 129% over the scheme using only PC.

We evaluated the hit ratio to show the superiority of the proposed scheme for Twitter, Weki-Talk, and YouTube datasets. A total of 5000 queries were executed, and the size of the cache was 2 GB. Figure 11 shows the evaluation results on hit ratio using each dataset. The proposed scheme showed the best hit ratio compared to that of the existing schemes, NYNN and CBGA. NYNN did not consider the characteristics of the graph with respect to data replacement inside the memory, and thus the lowest hit ratio was revealed in the evaluation using overall datasets. Although CBGA considered

the graph characteristics, it did not assign a weight to hot data inside the memory. Thus, it showed a lower hit ratio than that of the proposed scheme due to frequent data replacement. The performance evaluation results showed that the hit ratio of the proposed scheme was increased approximately by 223% compared with that of NYNN, and 129% compared with that of CBGA on average.

Figure 10. Hit ratio of single and two-level caches.

Figure 11. Hit ratio on each of the datasets.

In order to show the superiority of the proposed method, we measured the query processing time according to the amount of data. The amount of data ranged from 1000 to 10,000, and the size of the cache was 2 GB. Figure 12 shows the evaluation results on processing time between the proposed and existing schemes. The proposed scheme showed better processing time compared with those of NYNN and CBGA. The two existing schemes cached all neighbors in the memory, and thus data I/O was frequent. However, since the proposed scheme cached top-k data only, it showed a smaller processing time than those of the existing schemes. The performance evaluation results showed that

the processing time of the proposed scheme was improved on average by approximately 186% and 127% compared to those of NYNN and CBGA, respectively.

Figure 12. Processing time according to data request.

The query processing time was measured on the basis of the cache size to show the superiority of the proposed method. The data used were the Twitter data, and the data were 5000 in number. The size of cache was changed from 256 MB to 2 GB. Figure 13 shows the query processing time according to memory size. When the cache size was 256 MB, the proposed scheme showed lower performance than the existing scheme. The proposed method occupies a certain amount of memory space because it maintains a query management table to detect subgraph usage patterns. Therefore, the cache is reduced, and this does not sufficiently store the subgraph to be cached. In addition, the proposed method manages the cache space divided into UC and PC, and thus if the cache space is small, the UC and PC sizes are relatively small. Therefore, processing performance is deteriorated because it cannot cache a highly usable subgraph. However, as the cache size increases, the performance of the proposed scheme is higher.

Figure 13. Processing time according to memory size.

5. Conclusions

In this paper, we proposed a two-level caching strategy that caches subgraphs that are likely to be accessed in consideration of the usage pattern of subgraphs to prevent the caching of low-usage subgraphs and frequent subgraph replacement in the memory. Since frequently used subgraphs are likely used again, we assigned higher weights to them and they were cached in the used cache. Furthermore, neighbor data of recently used data were cached in the PC since they were likely to be used again. This paper also conducted the management of cached data in each of the caches and performed data replacement between cached and new data when memory was full. The excellence of the proposed scheme was proven through performance comparison with the existing schemes. The performance evaluation results showed that the proposed scheme exhibited better performance in terms of hit ratio and data processing time than the existing schemes.

Author Contributions: Conceptualization, K.B., S.Y., D.C., J.L., and J.Y.; methodology, K.B. and S.Y.; software, S.Y. and J.L.; validation, K.B., S.Y., and D.C.; formal analysis, K.B. and S.Y.; writing—original draft preparation, K.B. and S.Y.; writing—review and editing, J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (no. 2019R1A2C2084257, 2020R1F1A1075529); by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. NRF-2017M3C4A7069432); and by Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (no.B0101-15-0266, Development of High Performance Visual BigData Discovery Platform for Large-Scale Realtime Data Analysis).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ma, S.; Li, J.; Hu, C.; Lin, X.; Huai, J. Big graph search: Challenges and techniques. *Front. Comput. Sci.* **2016**, *10*, 387–398. [[CrossRef](#)]
2. Junghanns, M.; Petermann, A.; Neumann, M.; Rahm, E. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*; Springer: Cham, Switzerland, 2017; pp. 457–505.
3. Liu, S.; Wang, B.; Xu, M. SERGE: Successive Event Recommendation Based on Graph Entropy for Event-Based Social Networks. *IEEE Access* **2018**, *6*, 3020–3030. [[CrossRef](#)]
4. Klingström, T.; Plewczynski, D. Protein-protein interaction and pathway databases, a graphical review. *Brief. Bioinform.* **2011**, *12*, 702–713. [[CrossRef](#)] [[PubMed](#)]
5. Wang, P.; Chen, Y.; Lu, J.; Wang, Q.; Yu, X. Graphical Features of Functional Genes in Human Protein Interaction Network. *IEEE Trans. Biomed. Circuits Syst.* **2016**, *10*, 707–720. [[CrossRef](#)] [[PubMed](#)]
6. Shivraj, V.L.; Rajan, M.A.; Balamuralidhar, P. A graph theory based generic risk assessment framework for internet of things (IoT). In Proceedings of the International Conference on Advanced Networks and Telecommunications Systems, Bhubaneswar, India, 17–20 December 2017.
7. Edouard, A.; Cabrio, E.; Tonelli, S.; Thanh, N.L. Graph-based Event Extraction from Twitter. In Proceedings of the International Conference Recent Advances in Natural Language Processing, Varna, Bulgaria, 2–8 September 2017.
8. Namaki, M.H.; Lin, P.; Wu, Y. Event pattern discovery by keywords in graph streams. In Proceedings of the IEEE International Conference on Big Data, Boston, MA, USA, 11–14 December 2017.
9. Manzoor, E.A.; Milajerdi, S.M.; Akoglu, L. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.
10. Akoglu, L.; Tong, H.; Koutra, D. Graph based anomaly detection and description: A survey. *Data Min. Knowl. Discov.* **2015**, *29*, 626–688. [[CrossRef](#)]
11. Bok, L.; Kim, G.; Lim, J.; Yoo, J. Historical Graph Management in Dynamic Environments. *Electronics* **2020**, *9*, 895. [[CrossRef](#)]

12. Gonzalez, J.E.; Low, Y.; Gu, H. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Hollywood, CA, USA, 8–10 October 2012.
13. Zhu, X.; Chen, W.; Zheng, W.; Ma, X. Gemini: A Computation-Centric Distributed Graph Processing System. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016.
14. Gonzalez, J.E.; Xin, R.S.; Dave, A.; Crankshaw, D.; Franklin, M.J.; Stoica, I. GraphX: Graph Processing in a Distributed Dataflow Framework. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Broomfield, CO, USA, 6–8 October 2014.
15. Xu, Q.; Wang, X.; Li, J.; Zhang, Q.; Chai, L. Distributed Subgraph Matching on Big Knowledge Graphs Using Pregel. *IEEE Access* **2019**, *7*, 116453–116464. [[CrossRef](#)]
16. Shahrivari, S.; Jalili, S. Distributed discovery of frequent subgraphs of a network using MapReduce. *Computing* **2015**, *97*, 1101–1120. [[CrossRef](#)]
17. Choudhury, S.; Holder, L.B.; Chin, G.; Agarwal, K.; Feo, J. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In Proceedings of the International Conference on Extending Database Technology, Brussels, Belgium, 23–27 March 2015.
18. Semertzidis, K.; Pitoura, E. Top-k Durable Graph Pattern Queries on Temporal Graphs. *IEEE Trans. Knowl. Data Eng.* **2019**, *31*, 181–194. [[CrossRef](#)]
19. Ju, W.; Li, J.; Yu, W.; Zhang, R. iGraph: An incremental data processing system for dynamic graph. *Front. Comput. Sci.* **2016**, *10*, 462–476. [[CrossRef](#)]
20. Zhang, L.; Gao, J. Incremental Graph Pattern Matching Algorithm for Big Graph Data. *Sci. Program.* **2018**, *2018*, 1–8. [[CrossRef](#)]
21. Steer, B.A.; Cuadrado, F.; Clegg, R.G. Raphtory: Streaming analysis of distributed temporal graphs. *Future Gener. Comput. Syst.* **2020**, *102*, 453–464. [[CrossRef](#)]
22. Bok, K.; Jeong, J.; Choi, D.; Yoo, J. Detecting Incremental Frequent Subgraph Patterns in IoT Environments. *Sensors* **2018**, *18*, 4020. [[CrossRef](#)] [[PubMed](#)]
23. Wang, J.; Ntarmos, N.; Triantafillou, P. GraphCache: A Caching System for Graph Queries. In Proceedings of the International Conference on Extending Database Technology, Venice, Italy, 21–24 March 2017.
24. Liakos, P.; Papakonstantinou, K.; Delis, A. Realizing Memory-Optimized Distributed Graph Processing. *IEEE Trans. Knowl. Data Eng.* **2018**, *30*, 743–756. [[CrossRef](#)]
25. Dai, G.; Huang, T.; Chi, Y.; Zhao, J.; Sun, G.; Liu, Y.; Wang, Y.; Xie, Y.; Yang, H. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *38*, 640–653. [[CrossRef](#)]
26. Basak, A.; Li, S.; Hu, X.; Oh, S.M.; Xie, X.; Zhao, L.; Jiang, X.; Xie, Y. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In Proceedings of the International Symposium on High Performance Computer Architecture, Washington, DC, USA, 16–20 February 2019.
27. Yuan, Y.; Lian, X.; Chen, L.; Wang, G.; Yu, J.X.; Wang, Y.; Ma, Y. GCache: Neighborhood-Guided Graph Caching in a Distributed Environment. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 2463–2477. [[CrossRef](#)]
28. Ran, P.; Zhou, W.; Han, J. NYNN: An In-Memory Distributed Storage System for massive graph analysis. In Proceedings of the International Conference on Advanced Computational Intelligence, Wuyi, China, 27–29 March 2015.
29. Aksu, H.; Canim, M.; Chang, Y.; Korpeoglu, I.; Ulusoy, Ö. Graph Aware Caching Policy for Distributed Graph Stores. In Proceedings of the International Conference on Cloud Engineering, Tempe, AZ, USA, 9–13 March 2015.
30. Cicotti, P.; Carrington, L. A caching approach to reduce communication in graph search algorithms. In Proceedings of the International Workshop on Data Intensive Scalable Computing Systems, New Orleans, LA, USA, 16–21 November 2014.
31. Jiang, W.; Hu, H.B.; Xu, L.G. Query Acceleration of Graph Databases by ID Caching Technology. *J. Electron. Sci. Technol.* **2019**, *17*, 41–50.
32. Braun, P.; Cameron, J.J.; Cuzzocrea, A.; Jiang, F.; Leung, C.K. Effectively and Efficiently Mining Frequent Patterns from Dense Graph Streams on Disk. In Proceedings of the International Conference in Knowledge Based and Intelligent Information and Engineering Systems, Gdynia, Poland, 15–17 September 2014.

33. Iwabuchi, K.; Sallinen, S.; Pearce, R.A.; Essen, B.V.; Gokhale, M.B.; Matsuoka, S. Towards a Distributed Large-Scale Dynamic Graph Data Store. In Proceedings of the International Parallel and Distributed Processing Symposium Workshops, Chicago, IL, USA, 23–27 May 2016.
34. Zhu, X.; Serafini, M.; Ma, X.; Aboulmaga, A.; Chen, W.; Feng, G. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* **2020**, *13*, 1020–1034. [CrossRef]
35. Twitter. Available online: <http://socialcomputing.asu.edu> (accessed on 14 August 2017).
36. Weki-Talk. Available online: <https://snap.stanford.edu> (accessed on 17 August 2017).
37. Mislove, A.; Marcon, M.; Gummadi, K.P.; Druschel, P.; Bhattacharjee, B. Measurement and Analysis of Online Social Networks. In Proceedings of the ACM SIGCOMM Internet Measurement Conference, San Diego, CA, USA, 24–26 October 2007.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).