2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

GraphTinker: A High Performance Data Structure for Dynamic Graph Processing

Wole Jaiyeoba
Dept. of Computer Science, University of Virginia
Charlottesville VA, USA
oj2zf@virginia.edu

Kevin Skadron
Dept. of Computer Science, University of Virginia
Charlottesville VA, USA
skadron@virginia.edu

***Abstract*—Interest in high performance analytics for dynamic (constantly evolving) graphs has been on the rise in the last decade, especially due to the prevalence and rapid growth of social networks today. The current state-of-the art data structures for dynamic graph processing rely on the adjacency list model of edgeblocks in updating graphs. This model suffers from long probe distances when following edges, leading to poor update throughputs. Furthermore, both current graph processing models—the static model that requires reprocessing the entire graph after every batch update, and the incremental model in which only the affected subset of edges need to be processed—suffer drawbacks.**

In this paper, we present GraphTinker, a new, more scalable graph data structure for dynamic graphs. It uses a new hashing scheme to reduce probe distance and improve edge-update performance. It also better compacts edge data. These innovations improve performance for graph updates as well as graph analytics. In addition, we present a hybrid engine which improves the performance of dynamic graph processing by automatically selecting the most optimal execution model (static vs. incremental) for every iteration, surpassing the performance of both. Our evaluations of GraphTinker shows a throughput improvement of up to 3.3X compared to the state-of-the-art data structure (STINGER) when used for graph updates. GraphTinker also demonstrates a performance improvement of up to 10X over STINGER when used to run graph analytics algorithms. In addition, our hybrid engine demonstrates up to 2X improvement over the incremental-compute model and up to 3X improvement over the static model.

I. INTRODUCTION

The significant growth of social media today, and many other forms of Big Data that capture relationships among data points, require high-performance graph analytics. Furthermore, social network and knowledge graphs are evolving rapidly [34]. For example, Facebook users post an average of 37,000 likes and comments per second, Twitter records an average of 13,684 tweets per second [25], and Google's knowledge graph has grown from a trillion individual pages in 2008 to up to thirty trillion individual pages in 2013, which is a growth of about 30X in just five years. This has spurred the need for effective data structures for rapidly-evolving graphs, supporting efficient updates to the graphs and efficient real-time analytics.

Unfortunately, dynamic graphs present a number of added challenges for efficient graph processing. One major challenge is unpredictable edge-update patterns that may appear in dynamic graphs, which can result in poor update throughputs from the data structures. Another challenge is the need to support rapid growth or changes in the graph, which require a data structure that scales well.

In this paper, we present GraphTinker, a new data structure that efficiently supports dynamic graph processing. Our key goals are to (1) Minimize as much as possible the probe distance when following edges, and (2) Provide a more compact data structure representation. We achieve our first goal by two well-known hashing schemes - Robin Hood Hashing and Tree-Based Hashing - to form a balanced data structure that reduces probe distance in following edges, allows for higher load stability, and supports changes or growth in the graph. To achieve our second goal, we employ two compaction techniques. The first technique, which we term Scatter-Gather Hashing (SGH), allows for a closer packing of edge data belonging to vertices streamed into the data structure, and maintains this packing even as the graph grows larger. The second technique, which we term the Coarse Adjacency List (CAL) representation, allows us to maintain a compacted representation of the edges in the data structure.

Our first contribution improves the update throughput by up to 3.3X compared to the state-of-the-art graph data structure, STINGER, which relies on adjacency lists. At the same time, GraphTinker achieves higher load stability compared to STINGER (about 34% throughput degradation between the first and last input batch compared to about 72% experienced by STINGER). We also observe that Scatter-Gather Hashing (SGH) and the Coarse Adjacency List (CAL) representation dramatically improve the efficiency of the data structure (up to 91%) without the need for any form of pre-processing (making a pass over the graph to sort or compact the data structure to improve efficiency of the subsequent analysis).

We also show that, in addition to improving the efficiency of graph analytics with GraphTinker's more efficient data structure, a *hybrid* approach to dynamic graph processing further improves efficiency. Our hybrid approach dynamically chooses the most effective model *for each iteration*, choosing between the static approach, which requires reprocessing the entire graph after graph updates (as if the modified graph were an entire new, static graph), and the incremental model, which tries to update a prior analysis (e.g., connected components) using only the edges that were inserted or changed, but is less efficient that the static approach when the number of active vertices to be processed becomes very large. The hybrid approach is able to capture the best of both approaches.

II. BACKGROUND AND MOTIVATION

A. Graph Data Structure

Over the years, a number of data structures have been proposed for graph processing. There has been a tradeoff between the effectiveness of these data structures while updating edges versus supporting efficient real-time graph analytics. This is because more compacted graph data representations, which result in higher throughput performance, typically require large data movements when

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00110

performing updates. We explore prior data structures for dynamic graphs in this section.

Adjacency matrix: A classic data structure is an adjacency matrix, which holds a 2D matrix representation of the edges of the graph, so that an edge with endpoints u_i and v_i is located in the matrix position a_{ij} of the adjacency matrix structure. Even though this model provides O(*1*) edge insertion time, it is unsuitable for dynamic graph processing, because the overall sizes of today's graphs would warrant huge memory consumption $O(n^2)$ as well as very sparse representation of graph data.

Adjacency list: STINGER [6] is a state-of-the-art data structure for dynamic graph processing. It is a shared-memory data structure based on adjacency lists. STINGER's model consists of a *vertex table* and an *edge list*. Each element of the vertex table (called Logical Vertex Array) points to a given location in the edge list (Edge Block Array). The vertex table holds the vertices in the graph while the edge list consists of e*dgeblocks,* which hold the edges associated with each vertex. Edgeblocks can point to other edgeblocks to accommodate more space for edges belonging to a vertex. While this model allows some level of compaction of edges - meaning the edges in an edge block are packed close together - it still suffers from long probe distance during graph updates because entire chains of edgeblocks (belonging to a particular vertex) have to be traversed during an edge insertion or deletion process. This is because the edges belonging to the vertices are not sorted or hashed in any way and could be located in any of the edgeblocks belonging to a particular vertex. Additionally, this representation does not yield a highly compacted representation of edges in data structure because of the many non-contiguous edgeblocks that could be present in the database. These drawbacks direct our approach.

B. Graph Processing Models for Dynamic Graphs

Two primary execution models exist for updating analyses of a graph as the contents change.

Store-and-static-compute model: Traditional works [13-15] proposed graph processing on dynamic graphs by employing the store-and-static-compute model, in which updates are made to the graph in discrete time intervals and classic, static graph analytics algorithms are re-run on the entire graph after every update interval. If preprocessing is carried out before using this model to provide a compact representation of the graph data (e.g., from adjacency list to CSR representation), it can offer advantages because retrieving of edges can now be achieved in a highly contiguous manner. However, this model suffers from having to perform redundant computations, which can be significant when dealing with large graphs.

Incremental-compute model: Later works [20-24] explore an *incremental* approach to avoid this redundant computation. In this model, only regions affected by the batch update at discrete time intervals are processed. The vertices affected by the batch updates are referred to as *inconsistent vertices*, and they are identified as vertices in the graph whose properties change because of the update. These vertices become the first set of active vertices during graph processing. The advantage of this model is that it reduces the number of edges and vertices that must be recomputed and can lead to significant performance improvement when the reduction is substantial. However, this model incurs more expensive, non-contiguous data accesses to memory because the set of active vertices for a given iteration can be non-sequential and unsorted. In instances where many vertices are to be processed in a given iteration, the outcome using this model can be worse than the store-and-static-compute model.

The advantages each of these two graph computation models provide motivated us to create a hybrid model that leverages the benefits of both.

III. GRAPHTINKER OVERVIEW

As discussed in Section 2, one of the major disadvantages with the current state-of-the-art data structure (STINGER) for dynamic graph processing is the low update throughput realized due to long probe distance incurred while following edges. Probe distance in the context of performing edge updates refers to how many edges need to be traversed before the insertion or deletion of an edge is successfully completed. Longer probe distance decreases edge-update performance. This is because more memory accesses are required, and the amount of accesses can become significant with larger graphs, leading to poor load stability. Coupled with this limitation is the fact that STINGER's adjacency list structure does not provide sufficiently compact representation of edges in the database to allow for efficient graph engine computation. GraphTinker addresses the low probe distance issue by combining Robin Hood and Tree-Based Hashing. In order to solve the challenge of poor data compaction in edgeblocks, GraphTinker also maintains a 'Coarse' Adjacency List representation of edges in real-time, which enables a highly compact representation of the underlying data structure so that efficient graph processing can be performed without any form of pre-processing. This section explains in more detail the different algorithms and structures used in GraphTinker. STINGER was implemented in C, while we implement GraphTinker in C++.

A. Robin Hood Hashing

The Robin Hood Hashing (RHH) algorithm [3-5] provides a solution for achieving low probe distance. A brief description of how the algorithm works is discussed below.

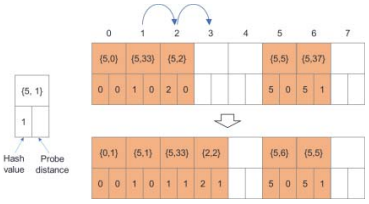

Fig. 1: Robin Hood Hashing

Fig 1 shows the insertion process of an edge into a simple hash table using the Robin Hood Hashing algorithm. Assume {i, j} is an edge between a vertex *i* and *j*. If this edge is to be inserted into the hash table, a suitable hash function (e.g., $h = V_{id} \bmod C$, where V_{id} is ID of the edge's source vertex and C is the capacity of the hash table) is used to compute the initial position **(**known as *initial bucket***)** of

the hash table from where inspection begins. For the example in Fig. 1, the initial bucket for the edge x is 1. Next, because the edge entry in bucket 1 of the hash table is already occupied, the probe distance of both edges are compared to decide who now stays in the bucket. If the probe distance of the edge currently present in the bucket is lower than that of the edge to be inserted, then the edge to be inserted now swaps the edge currently present in the bucket, so that the edge formerly in the bucket now becomes the new floating edge looking for another location to be inserted. Otherwise, the edge to be inserted skips over that bucket to inspect the bucket. Probe distance, in the context of Robin Hood Hashing, refers to the distance between the original hashed positions of the edge to its current displaced position. For the example, the new edge (1, 5) wins because its probe distance is equal to the probe distance of the edge already present in bucket 1 (1, 6). Since edge (1, 6) is evicted, it must find a new bucket to occupy. When comparing with edge (2, 2), it wins because it is "poorer" than the one in bucket 2 (2, 2) – meaning its probe distance is greater than the one in bucket 2 (2, 2). Edge 2, now evicted, finds bucket 3 empty and moves there.

B. GraphTinker Data Structure

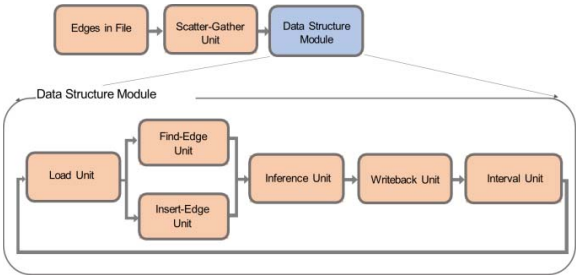

Fig. 2: GraphTinker components

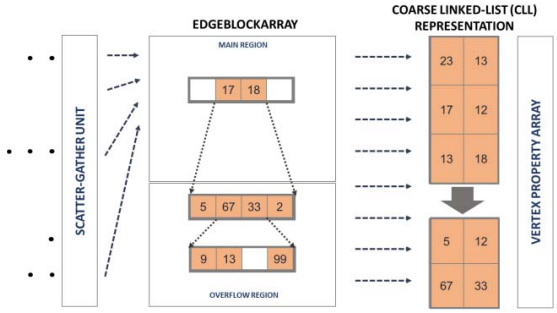

Fig. 3: Different structures contained in GraphTinker

GraphTinker's major structures are the *EdgeblockArray*, (consisting of *Subblocks* & *Workblocks*), the *VertexPropertyArray,* the *Scatter-Gather Hashing Unit (SGH unit)* and the *Coarse Adjacency List Array* (*CAL Array*). Fig. 3 illustrates how these different structures are connected together in GraphTinker. We shall first describe these, and then explain how they achieve GraphTinker's overall purpose.

The EdgeblockArray: As explained earlier, a fundamental goal of GraphTinker is to allow for decreased probe distance while following edges. The key property of the EdgeblockArray is that it provides support for the RHH algorithm as well as support to expand arbitrarily to accommodate more edges, while still maintaining an average low probe distance. As shown in Fig. 4, the EdgeblockArray is composed of row substructures called *edgeblocks*, each having a default length (PAGEWIDTH), which is where the edges of the graph are stored. The vertex that owns these edges is stored in the VertexPropertyArray. Every edgeblock in the EdgeblockArray is divided into smaller sections called *Subblocks,* which form the basis of the EdgeblockArray expansion. These Subblocks are themselves divided into smaller sections called *Workblocks***,** which house the most primitive unit of the EdgeblockArray (called edge-cells). An edge-cell may or may not contain an edge.

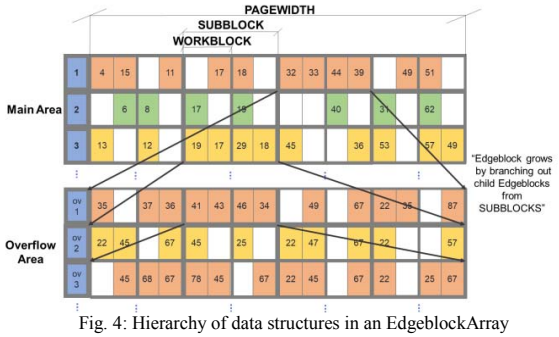

Fig. 4: Hierarchy of data structures in an EdgeblockArray

An EdgeblockArray is divided into two parts, namely the *main region* and the *overflow region*. The main region is composed of edgeblocks (called *top-parent* edgeblocks), which store edges of vertices, and is indexed by the vertex source IDs. Therefore, every index of the main region of the EdgeblockArray consists of edges belonging to a source vertex. The overflow region is composed of edgeblocks which have the same property as the edgeblocks in the main area, but are descendants of Subblock sections of either the main region (1st generation descendant) or of the overflow region (>1st generation descendant) itself.

The overflow region enables the underlying dynamic graph to grow arbitrarily. When the Subblock regions (of edgeblocks either in the main area or overflow area) become congested and filled with edges (in accordance with the Tree-Based Hashing algorithm), they "branch out" into child edgeblocks, which reside in the overflow area. Because these child edgeblocks possess the same characteristics as the parent edgeblocks, they themselves are also divided into Subblocks, which can also "branch out" when congested. Fig. 4 gives a simple example of how and when this branching out process occurs. As shown, the third Subblock belonging to vertex with ID one *(v1)* was at some point congested and branched out to form a new child edgeblock *(ov1)*. Sometime during edge updates, the second Subblock of this child edgeblock *(ov1)* also became congested with edges, and in turn, branched out to form its own child edgeblock *(ov3)*. With this descendant-level arrangement of edgeblocks, the average probe distance when following edges of a particular vertex v_i is of the order *O(log(n))* as compared to the adjacency list representation which is *O(n)* where n is the degree of the v_i. Therefore, with graphs having much larger average degrees, performance degradation is smaller using GraphTinker than with the adjacency list based data structures.

Tree-Based Hashing: The Tree-Based Hashing scheme is responsible for the "branch out" process that occurs whenever more edgeblocks need to be allocated to accommodate a new edge (during insertions), or when more edgeblocks belonging to a particular vertex need to be searched for the new edge (during deletions). Whenever a Subblock is congested, the Tree-Based Hashing algorithm allocates an empty (if not already existing) edgeblock in EdgeblockArray, and points to it from the Subblock. It also calculates the new Subblock location in the child edgeblock to continue the insert or delete process. This decision is based on a user-defined hash function and the parameters to this function are the default PAGEWIDTH of an edgeblock and the *destination vertex ID* of the new edge.

Subblock regions: The Subblock regions are components that make up edgeblocks in an EdgeblockArray. The *Subblock* region is the first layer of granularity of the edgeblock (in the EdgeblockArray). It represents the component of the EdgeblockArray which is capable of 'branching out' (when congested) into *child edgeblocks* (located in the overflow region) in order to house more out-edges for a particular source vertex. It simply achieves this by pointing to its child edgeblock which is located in the overflow region.

Workblock regions: The *Workblock* region, on the other hand, forms the second layer of granularity of the EdgeblockArray. Its main purpose is to parameterize the granularity at which edge data are retrieved from the EdgeblockArray for inspection. During the process of updating a new edge, when the Tree-Based Hashing scheme allocates a (or retrieves an existing) Subblock to be retrieved from the EdgeblockArray, the Subblock is retrieved one Workblock at a time for the RHH process. Therefore, having too large Workblock sizes would increase the probability of a successful completion of the RHH process in that retrieval, but at the same time would increase the number of edges retrieved from DRAM. Therefore, the concept of having Workblock size as a parameter during configuration of GraphTinker allows the user to select an optimum performance point.

The VertexPropertyArray: The VertexPropertyArray is the array structure which houses the properties of the vertices of the graph. It stores information pertaining to the vertices such as the degree, value and any flags associated with it and indexed by the vertex IDs of the vertices. This structure works with the EdgeblockArray during the graph computation process because both edge data and vertex properties need to be retrieved for computation.

Scatter-Gather Hashing: Among the goals of GraphTinker is to provide a highly compacted graph data representation such that no form of pre-processing is necessary in order to do highly efficient graph analytics in real-time. However, edges streaming into the data structure unit can be highly non-contiguous (by their source and destination vertex ids). This can result in a sparse representation of edge data in the EdgeblockArray. For example, if two edges streaming into the EdgeblockArray have source vertex ids 34 and 22789, it means the *top-edgeblocks* belonging to them would be located at index locations in the EdgeblockArray which are 22755 indexes apart. This means unless some form of pre-processing is carried out, graph processing of this data may involve traversing all vertices in the data structure when seeking to process all edges. And at the early stages of a graph's life, where not many edges have been loaded compared to the full capacity of the graph, this can result in significant performance degradation. Therefore, in to ensure that only "non-empty" vertices (i.e., vertices with degrees greater than zero) are traversed at any time during graph processing, we incorporate a Scatter-Gather Hashing (SGH) function (consisting of a function and a hash table), which interfaces to the EdgeblockArray. This forms the first level of compaction of data supported by GraphTinker (the second of which is the Coarse Adjacency List representation, discussed shortly). For every new edge to be updated, the source vertex ID of that edge is inspected. If the source vertex ID has not been hashed before, it is hashed by the Scatter-Gather Hashing function to obtain a new source vertex ID associated with it. This Hashing is simply obtaining the next unused index location in the EdgeblockArray starting from zero. On the other hand, if the source vertex ID has been hashed before, the Scatter-Gather Hashing table is checked to obtain the formerly hashed id. In either of both situations, the edge is now associated with a new *hashed* source vertex ID which is now used by the data structure to perform the update operation. The mapping between the original source vertex ID and the new hashed source vertex ID (and vice versa) is maintained by the Scatter-Gather Hashing table. With this hashing scheme, it means the new *hashed* source vertex ID now becomes the index location (of the EdgeblockArray) of the *top-edgeblock* of that source vertex. With this functionality, the SGH technique helps to maintain a list of non-empty vertices (i.e., vertices owning edges) of the graph at any stage of its lifetime. This way, during graph processing, not all the vertices need to be inspected while retrieving all updated edges from the EdgeblockArray, and performance can be improved.

'Coarse' Adjacency List EdgeblockArray:

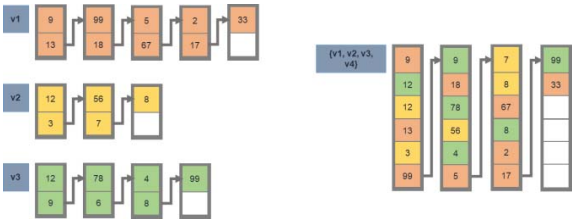

Fig. 5: logical representation of regular adjacency list representation of edges (left) vs Coarse Adjacency List representation of edges (right).

Besides the effectiveness of GraphTinker in providing high performance for edge insertions and deletions, there is a need for it to be capable of performing high performance analytics of dynamic graphs on-the-fly (i.e., without any form of pre-processing required). This facilitates the need for additionally maintaining a highly-compacted form of edge data representation to allow for highly-contiguous graph data accesses from DRAM which helps improve graph processing performance. State-of-the-art dynamic data structure models (such as STINGER) are inefficient in addressing this problem because the Adjacency list representation of edges does not provide high-enough edge

data compaction due to the many non-contiguous edgeblocks locations in memory within a vertex and across a given number of vertices (as discussed earlier in section II). GraphTinker takes the idea of edge data compaction to a higher level with the Coarse Adjacency List (CAL) strategy.

To this effect, we maintain a separate, 'Coarse' Adjacency List representation copy of the edges in database (which we call the Coarse Adjacency List (CAL) EdgeblockArray). The Coarse Adjacency List representation is quite similar to STINGER's adjacency list style representation, but with the notable difference that in the CAL representation, several source vertices share an entry. This means that in the CAL data structure, edges are stored in blocks, each of which contains metadata with a pointer pointing to both the previous and next block housing the same group of edges as it contains. The logical illustration of this data structure is as shown in Fig. 5. To form the CAL EdgeblockArray, source vertices are partitioned into different *groups* according to their vertex ids, and group ids are assigned to each group, where each group represents a given contiguous range of source vertex IDs. For example, if every group consists of 1024 vertices, then source vertex ids from 0 to 1023 all belong to group 0, etc. Each group id now represents an index in an adjacency list table (the Coarse Adjacency List). Just as with STINGER (which relies on the adjacency list representation), edgeblocks (containing edges) are assigned to each group id and expansion of edgeblocks belonging to a group is done by more edgeblocks. However, unlike with STINGER, the edges in an edgeblock can belong to different vertices in a group. Therefore, each edge in an edgeblock of the CAL EdgeblockArray also maintains a reference to its source vertex.

The process of inserting, updating or deleting an edge happens while updating the EdgeblockArray. Whenever a new edge is inserted in the EdgeblockArray during a graph update, its source vertex id is inspected to find the group that vertex belongs to. Then, a Logical Vertex Array is looked up to find the last assigned edgeblock to this group and the last unoccupied edge slot in this edgeblock. This edge copy is then inserted into this empty slot. In order for the original edge (in EdgeblockArray) to maintain the location of the edge copy, a pointer (called the edge's *CAL-pointer*) is maintained by this original edge pointing to the edge copy. This pointer points to the location of the edge copy in the Coarse Adjacency List (CAL EdgeblockArray). Therefore, during updating in the EdgeblockArray, its copy in the Coarse Adjacency List is also updated via the edge's *CAL-pointer* information. A similar process occurs when deleting an existing edge in the EdgeblockArray. Here the *edge's CAL-pointer* is retrieved and the copy of that edge is deleted (flagged as invalid). Because this process of updating the CAL EdgeblockArray does not involve traversing edges, its overhead to GraphTinker's operation is minimal. In essence, CAL representation provides a more compact edge data representation in database and therefore reduces the number of non-contiguous edge data accesses from memory during graph analytics computation.

GraphTinker's Interface Components: Fig. 2 illustrates the different components that connect together to allow the various operations supported by GraphTinker – the *Scatter-Gather Hashing unit*, the *load unit*, the *find-edge unit*, the *insert-edge unit*, the *inference unit*, the *writeback unit* and the *interval unit*. The load unit, the writeback unit, and the Scatter-Gather Hashing unit directly interface GraphTinker, while the rest manage the different operations supported by GraphTinker. The *load unit* is responsible for using the information contained in the next incoming edge to retrieve the relevant *Workblock*s from the EdgeblockArray for RHH. The *find* and *insert unit* implement the FIND and UPDATE modes of the data structure respectively, so that it is capable of performing deletes, inserts and updates to edge batches via the RHH algorithm. The *inference* and *interval units* maintain appropriate control flow of Workblock retrievals belonging to the vertex being investigated, while the writeback unit writes back processed Workblocks back to the EdgeblockArray.

C. GraphTinker's Basic Operations

GraphTinker is designed to support high-speed updates to graphs in the underlying database, while also supporting high-speed traversal during graph analytics. It supports functionalities such as edge insertions, edge deletions, as well as edge retrievals for graph analytics computations - such as breadth-first-search (BFS), single-source-shortest-path (SSSP) and connected-components (CC) algorithms. These functionalities are discussed below.

Inserting a new edge: For each edge insertion, GraphTinker allows two modes: the FIND mode and the INSERT mode. The FIND stage attempts to search for the edge belonging to the associated vertex in the EdgeblockArray, while the INSERT stage attempts to insert a new edge if the FIND stage is unsuccessful. In cases where deletions have previously been made and an empty slot is created, the INSERT stage can also insert edges into these empty slots. For each stage, the edge is hashed (based on its source and destination vertex ids) to determine the appropriate Subblock location in the EdgeblockArray to start looking from. The data structure then retrieves Workblocks belonging to this Subblock in sequential order, running the find/RHH algorithm on each Workblock depending on the current operation mode. If the update is still unsuccessful after all the Workblocks of a Subblock are inspected, newer 'branches' (edgeblocks) are created (if not already available) out of the Subblock, rehashing is done again, and the same process continues in the newly-hashed child Subblock region. This process can continue for a few generations for very large-degree vertices. "Branching out" from a Subblock to form child edgeblocks is simply achieved by inserting a pointer into the Subblock, pointing to the newly created child edgeblock.

Deleting an edge: Two edge-deletion mechanisms are incorporated in GraphTinker: delete-only and delete-and-compact. The delete-only mechanism is straightforward; instead of erasing all data (key, value and probe distance) of the element and moving succeeding elements forward, a flag (tombstone) is set to indicate that no edge exists in the bucket anymore. Therefore, any edge that traverses this bucket location the next time sees this bucket as vacant. The delete-and-compact mechanism, on the other hand, attempts to compact the data structure whenever a deletion is made, reducing probe distance and freeing edgeblocks for

subsequent insertions and maintaining compactness of the database. It achieves this by deleting edges (flagging as tombstone) from appropriate child edgeblocks of the data structure. These edges removed are then inserted into the slots where the deletions took place. By doing this, the holes in between edges in an edgeblock that are created by deletion of edges are filled up during the deletion process, thus allowing the data structure to stay compact even as more and more edges are deleted from the database. In order to avoid the significant overhead and complexity of edge tracking associated with the swapping process of the RHH algorithm, only the Tree-Based Hashing algorithm is enabled with this mechanism, with the RHH algorithm turned off.

Retrieval of edges for dynamic graph analytics: GraphTinker also allows for retrieval of edge streams for analytics on dynamic graphs. Retrieval of edges via GraphTinker can be either from the EdgeblockArray or from the CAL EdgeblockArray, depending on whether the mode of execution is the full or incremental processing mode. More details on these different modes are discussed in section IV.B.

D. GraphTinker on Multicore Processing Architectures

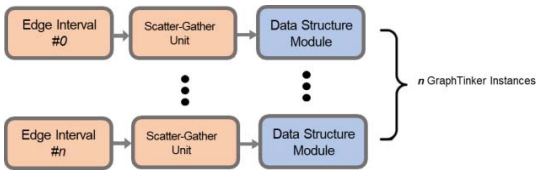

Fig. 6: Parallel Instances of GraphTinker

Parallelization: In order to leverage multiple cores in GraphTinker, we exploit the available parallelism that exists between vertex updates during GraphTinker update operations. Therefore, we parallelize GraphTinker across multiple cores/systems by partitioning the edge dataset into several ***intervals*** according to where their source vertex ids hash to, and then by loading each interval into separate GraphTinker instances, as shown in Fig. 6.

IV. HYBRID GRAPH ENGINE

As mentioned in Sec. I, we observe that a hybrid approach that chooses the most effective mode between the store-and-static-computation mode and the incremental computation mode *for every iteration,* further improves performance. This section discusses the graph computation model which we employ in our hybrid graph engine, as well as our proposed hybrid engine model.

A. Graph Computation Model

The Edge-Centric GAS model: The Gather-Apply-Scatter (GAS) model has gained wide popularity over the past years because it simplifies and effectively expresses a broad range of graph algorithms (e.g., heat simulation algorithms, connected components, and breadth-first search algorithms). It is used as our graph engine model because previous works [26] have demonstrated the flexibility of the GAS model in running a wide variety of graph algorithms. In the GAS model, the input data can be described as a directed graph, G = (*V, E*), where *V* denotes the vertex set and *E* denotes the edge set. A GAS computation iterates over three sequential phases, namely Gather, Apply and Scatter phases. In the Gather phase, incoming messages are processed and combined into one message. In the Apply phase, vertices use the combined message to update their respective states, while in the Scatter phase, vertices can send messages to their neighbors. The GAS model can be employed using *vertex centric* (*VC*) or *edge centric* (*EC*) approaches, depending on whether the Scatter and Gather phases iterate over and update edges or vertices.

The *EC* model, which we use in our graph engine implementation. This model allows users to express their graph algorithms in two alternating computation phases: the processing phase and the apply phase.

Processing Phase: In the processing phase, all outgoing edges from every active vertex are processed according to a given user-defined function (*processEdge*), and the results of these computations are used to *reduce* the vertex property of the associated vertices. The reduction modifies the old vertex property according to a user-specified function. This newly-generated vertex property is then buffered in a temporary buffer (*VTempProperty* Array) until this phase is complete.

Apply Phase: In this phase, the newly-generated vertex properties (which were stored in the *VTempProperty* Array) are then committed to the original vertex array (*VPropertyArray*). The commit process occurs according to a user-defined function (*apply*), and this function generates a new set of active vertices to be used in the processing phase of the next iteration.

A graph algorithm which is conformable to the edge-centric paradigm only needs have separate definitions for its *processEdge*, *reduce* and *apply* functions, and all other functionalities of the graph engine can remain unaltered. Future work on GraphTinker will explore the efficiency of the vertex-centric model with our data structure.

B. Our Hybrid Graph Engine Model

Our proposed hybrid graph engine is designed to incorporate the best of both worlds of two different dynamic graph computation models (namely the store-and-static-compute model and the incremental-compute model) for every iteration of the graph computation process. This model is suited for algorithms such as BFS, SSSP, and CC, where not all vertices need to be active in every iteration. (Otherwise, incremental processing is not an option.) In this section, we shall discuss the different modes that exist in our hybrid graph engine model and how these modes can be multiplexed to achieve better performance:

By deciding between full and incremental processing modes at every iteration, we can select the best execution path for the next iteration. Our hybrid engine gathers certain information during the apply phase of every iteration *(i)* and uses this information to predict the most optimal computation mode (whether full or incremental processing mode) to be carried out in the processing phase of the next iteration *(i+1)*. A typical scenario where this mode shows superior performance is in situations where the current iteration *(i)* has only a few active vertices it needs to process in the processing phase, but the apply phase computation

indicates that the processing phase of the next iteration *(i+1)* would have a large number of active vertices that would be processed. In this scenario, the current iteration *(i)* might best benefit from the incremental processing (IP) mode while the next iteration *(i+1)* might best benefit from the full processing (FP) mode.

In our current version, the hybrid engine collects information in the apply phase of every iteration, namely the number of active vertices for the next iteration, the total degrees of all active vertices for the next iteration, the current size of the graph, and the maximum size attainable by the evolving graph. A prediction value is calculated using by using the prediction formula:

$$mode(i+1) = \begin{cases} FP, & T > threshold \\ IP, & T < threshold \end{cases}$$

Where:

$$T = \frac{A}{E}$$

$$threshold = 0.02$$

FP: Full processing mode
IP: Incremental processing mode
A: Total number of active vertices for iteration $i+1$
E: Total number of edges loaded so far
mode(i): mode registered for iteration i

As shown from the formula above, the decision on whether full or incremental processing mode should be chosen for the next iteration depends on an estimated measure of what fraction of edges would need to be processed in the next iteration. Separate experiments were run for both full and incremental processing modes to investigate the tradeoffs between sequential streaming versus random retrieval of edges from the graph structure, and how these vary with the number of edges retrieved. This resulted in the choice of an optimal threshold value of 0.02. Future work would aim to factor in other heuristics such as number of degrees of the active vertices and memory characteristics in order to attain higher predictive accuracy of the prediction formula.

C. Implementation of the Hybrid Graph Engine

Fig. 7 shows the component-level arrangement of our hybrid engine model, and how the different parts are connected together to achieve its overall functionality.

Set Inconsistency Vertices unit: There is a need to set the initial inconsistency vertices after every batch update step and before the graph processing process is started. Inconsistency vertices are vertices in the graph whose properties change because of the update. These initial inconsistency vertices become the active vertices for the first iteration.

The implementation of this module differs slightly depending on the algorithm to be implemented but only requires slight adjustments depending on the algorithm to be implemented. For example, in the BFS algorithm, the vertices affected by the update batch comprise the source vertices of the edges in the update batch, while the inconsistency vertices when running Weakly Connected Component (CC) comprise both the source and destination vertices of the edges in the update batch. During configuration of GraphTinker, this unit is automatically generated depending on the algorithm to be run.

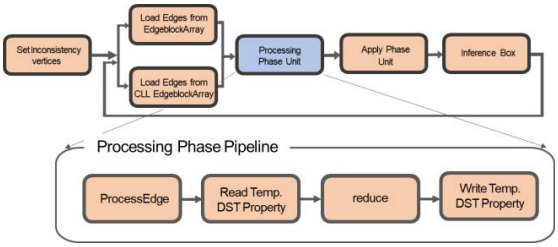

Fig. 7: Hybrid Engine Implementation

LoadEdges unit: This unit loads edges from the EdgeblockArray structures. Implementation of this unit differs between the full and the incremental processing mode. When the hybrid graph engine is in full processing mode, the edges are loaded from the CAL EdgeblockArray, because it provides a very compacted data representation of the graph edges. On the other hand, when the hybrid graph engine is in the incremental processing mode, edges are loaded from the EdgeblockArray because the accesses are from non-contiguous vertex index locations.

Graph processing pipeline: This unit implements the computation pipeline for the Edge-Centric graph processing model described in the processing phase in IV.A.

Apply unit: This unit executes the apply phase of the graph processing model as described in Sec. IV.A.

Inference Box Unit: The inference box unit is responsible for deciding, based on the user-defined heuristics, what the next execution mode should be for the next iteration - i.e., full or incremental processing mode.

V. EVALUATION

In this section, we experimentally evaluate the performance of GraphTinker as a standalone data structure (evaluating on update throughput) and as a basis to run graph algorithms (evaluating on the performance of these algorithms). We also evaluate the performance of our Hybrid Graph Engine model.

A. Experimental Setup

Implementation: We implemented GraphTinker on an Intel CPU (Intel® Xeon® E5-2620 v4) with 8 physical cores, operating at 2.10GHz, with 512GB DRAM. All experiments were run on the CPU. The PAGEWIDTH, Subblock and Workblock sizes of GraphTinker were chosen to be 64, 8 and 4 respectively because our experiments found that they define a good balance between effective data structure performance in updating edges and in graph analytics computation. We compare GraphTinker to the previous state-of-the-art data structure for dynamic graph processing, STINGER [6]. STINGER is a shared memory (in core) parallel dynamic graph processing framework. It performs updates about 3 times faster compared with 12 open-source graph databases and libraries [17], such as Boost Graph Library, DEX, Giraph and SQLite. We used version 15.10, available on GitHub. STINGER's configuration was set to have an average edgeblock size of 16. The batch size of edges used in the experiments to compare GraphTinker and STINGER is 1 million edges per batch. The choice on batch size does not have any impact on

results.

Datasets: We evaluate the performance of GraphTinker (insertions and deletions) using a mix of both synthetic and real-world graph datasets. The synthetic datasets are generated from the Graph500 RMAT generator [2], while the real-world datasets are obtained from the University of Florida's Sparse Matrix Collection [8]. The properties of these datasets are shown in Table 1.

Graph dataset	Type	#Vertices	#Edges
RMAT_1M_10M	synthetic	1,000,192	10,000,000
RMAT_500K_8M	synthetic	524,288	8,380,000
RMAT_1M_16M	synthetic	1,048,576	15,700,000
RMAT_2M_32M	synthetic	2,097,152	31,770,000
Hollywood-2009	real world	1,139,906	113,891,327
Kron_g500-logn21	real world	2,097,153	182,082,942

Table 1. Graph datasets under evaluation

Benchmark algorithms: We evaluate the performance of GraphTinker as an efficient data structure for dynamic graph processing by evaluating it in conjunction with several algorithms: breadth-first-search (BFS), single-source-shortest-path (SSSP), and connected-components (CC). These algorithms were chosen because they could be modelled using both full and incremental processing modes, and because they are important algorithms in the graph community.

B. Result Summary

GraphTinker vs. STINGER (Insertion throughput performance)

Fig. 8 shows the insertion throughput of GraphTinker and STINGER's data structures (i.e., without any graph analytics computation taking place) when used to insert edges into the graph using the Hollywood2009 dataset. Two different setups for GraphTinker were used: First, when the GraphTinker is used with the CAL feature and second, when used without it. A single thread was used to run the experiment. The y-axis in the figure represents the insertion throughput (in million edges per second) while the x-axis represents the input sizes of edges loaded (in million edges inserted in batches of 1 million edges per batch).

As shown in Fig. 8, GraphTinker's insertion throughput outperforms STINGER by up to 2.7X when GraphTinker is used with the CAL module and up to 3.3X when GraphTinker is used without the CAL module. One important observation from the plot is that GraphTinker shows less performance degradation than STINGER as the load (input size) increases. As shown, GraphTinker decreased from 1.6 million edges/sec in the fifth input batch to 1 million edges/sec in the last batch, giving about 34% throughput degradation, while STINGER decreased from 1.3 million edges/sec in the fifth input batch to 0.4 million edges/sec in the last input batch, giving about 72% throughput degradation. This implies that GraphTinker has better load stability compared to STINGER. The improvement in throughput and the higher load stability of GraphTinker over the STINGER structure is because GraphTinker makes fewer edge traversals during updating. This makes GraphTinker more efficient in DRAM accesses than STINGER.

Fig. 9, on the other hand shows the throughput performance of GraphTinker and STINGER for insertions using different datasets. As shown, GraphTinker outperforms STINGER across all the datasets. One interesting thing to note from the figure is that as the size of the datasets increases, GraphTinker's performance advantage also increases. This is again because STINGER has to do more edge-traversals than GraphTinker.

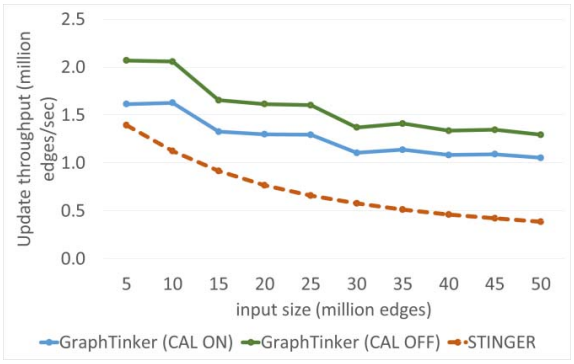

Fig 8: Insertion throughput for GraphTinker vs. STINGER with different input sizes and using the hollywood-2009 dataset

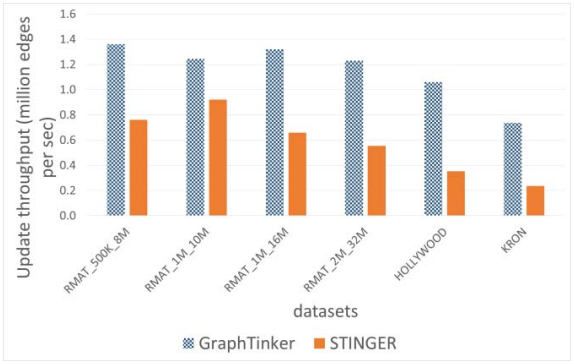

Fig. 9: Insertion throughput for GraphTinker vs. STINGER on different datasets and with batch size of 1 million edges

GraphTinker vs. STINGER (multicore performance)

Fig. 10 shows the performance of the GraphTinker data structure as the number of cores of the multicore CPU system increases. The y-axis in the figure represents the insertion throughput (in million edges/sec) of the data structure, while the x-axis shows the number of cores used. The dataset used in this experiment is the hollywood2009 dataset. GraphTinker maintains its performance advantage as core count increases. As shown, GraphTinker outperforms STINGER in each case. In this experiment, we observe that, for each of the different number of cores used, STINGER starts off with fairly good insertion throughputs during the first set of batch insertions, but then experiences rapid deterioration as more batches are inserted. For example with 8 cores, STINGER experienced a decrease from about 3.4 million edges/sec in the first iteration to about 1 million edges/sec in the last iteration. In contrast, with GraphTinker, we observe far less degradation in throughput as subsequent batches are inserted.

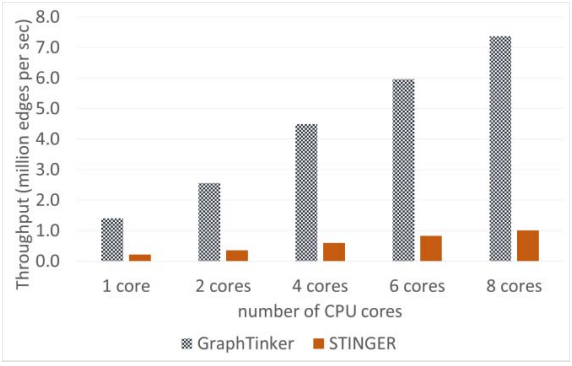

Fig. 10: Update throughput for GraphTinker vs. STINGER using different number of CPU cores

GraphTinker vs. STINGER for Different Graph benchmarks

In order to evaluate the impact on performance of GraphTinker on important graph algorithms, we ran graph analytics on BFS, SSSP, and CC using GraphTinker as the data structure and our hybrid engine as the graph engine. In each experiment, edges of the given dataset are loaded into the data structure in batches (of 1 million edges) to update the graph. After each batch insertion, the graph engine runs the given graph analytics algorithm on the current state of the graph. This two-step process continues in turn until the final batch of edges is loaded and there are no more edges remaining to load. For comparison, we also ran graph analytics using STINGER.

Figs. 11, 12 and 13 show the performances of BFS, SSSP and CC algorithms when using GraphTinker and STINGER. The y-axes in the figure represent the throughput (in million edges/sec) when running each algorithm across the different datasets, while the x-axes represent the different datasets used in the experiments.

As shown, when the hybrid engine is configured to run in full processing mode and using GraphTinker as the data structure, it demonstrates significantly better performance than when using STINGER (up to 10X performance improvement).

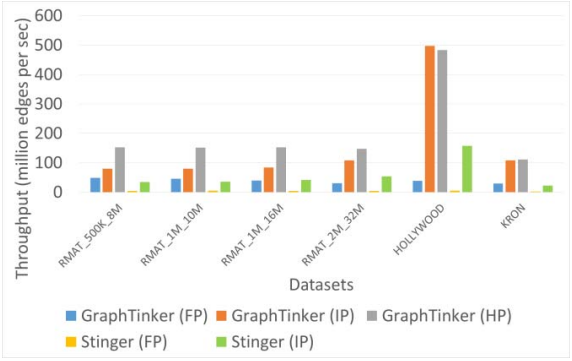

Fig. 11: Processing throughput for GraphTinker vs. STINGER when running BFS on different datasets

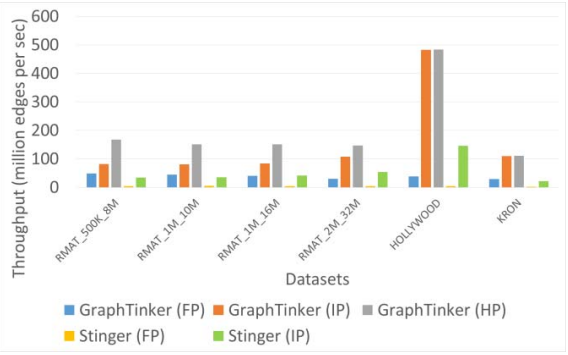

Fig. 12: Processing throughput for GraphTinker vs. STINGER when running SSSP on different datasets

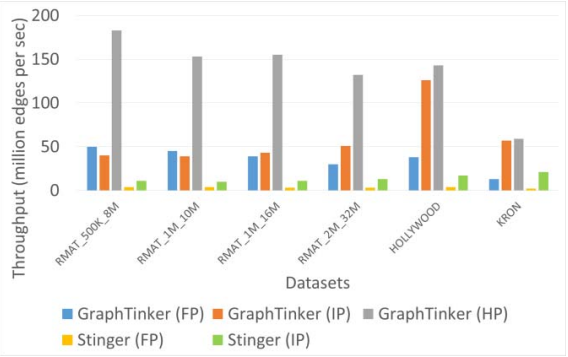

Fig. 13: Processing throughput for GraphTinker vs. STINGER when running CC on different datasets

There are two major reasons for this. First, the Coarse Adjacency List (CAL) EdgeblockArray representation of edges maintained by GraphTinker allows a highly-compacted representation of edges, which reduces non-contiguous memory accesses compared to STINGER. Second, the Scatter-Gather Hashing (SGH) scheme implemented in GraphTinker allows it to reduce DRAM memory accesses during edge retrievals compared to STINGER. We conducted experiments where we disabled the CAL and the SGH features of GraphTinker and observed that GraphTinker then results in only about 1.5 times better than STINGER when running in full processing mode. Additional experiments show that the SGH and CAL feature account for a combined improvement of over 91% in GraphTinker's performance when enabled. This shows how significant these two features are to GraphTinker as an effective data structure for analytics of dynamic graphs.

These figures show that the hybrid mode always demonstrates better performance than the full and incremental processing modes for all the algorithms (BFS, SSSP and CC), and using all the datasets. This is because the inference box (the decision-maker) of the hybrid engine makes excellent predictions on the best execution path to take for every iteration. The reason for the significant gap in performance with the hybrid engine mode over both full and incremental processing modes in some cases (such as in CC) is that, in these instances, the hybrid engine makes especially successful predictions (we observed up to 97% correctness).

Figs. 11, 12 and 13 also show that there are instances

where the incremental processing model can perform worse than the full processing model. An example is the CC experiment on dataset RMAT_500K_8M. In this experiment, about 30% of the iterations involved a very large number of active vertices (> 100,000 active vertices). This caused IP to sometimes perform worse than FP, with ranges of ~3X to 9X performance degradation in some iterations. This performance degradation is caused by the significant amount of non-contiguous memory accesses incurred by IP in these scenarios. However, in many applications and data sets, IP is superior. Hybrid execution is thus an important ingredient of an efficient graph engine.

Comparison of GraphTinker and STINGER edge deletion mechanisms

We evaluate the impact of GraphTinker's deletion mechanisms on data structure and graph analytics performance. We run the BFS algorithm (in FP mode) on the RMAT_2M_32M dataset using a single core. The graph is initially fully loaded, after which deletions are then made (at 1 million edges per batch) in batches until the database is empty. This is to evaluate GraphTinker's performance as deletions are made to the data structure. Also, graph analytics is performed after every batch is deleted in order to evaluate GraphTinker's performance in analytics (i.e., after edge deletions are carried out). The performance of GraphTinker for edge deletions and also for analytics after edge deletions have been made is observed. For comparison, we compare to STINGER.

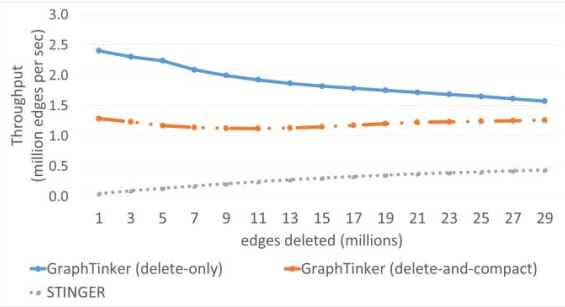

Fig. 14: Edge deletions throughput for GraphTinker vs. STINGER data structure with different input sizes and using the RMAT_2M_32M dataset

Fig. 14 shows the throughput of both data structures (GraphTinker and STINGER) from the experiment when used for edge updates (deletions) only and without any graph analytics process taking place. The y-axis in the figure represents the deletion throughput (in million edges per second) while the x-axis represents the amount of edges deleted (in million edges). As shown, the delete-only mechanism for GraphTinker outperforms the delete-and-compact mechanism by up to 2X when the first batch is deleted and only about 1.2X when the last batch is deleted from the database. Both deletion mechanisms, however, outperform STINGER's deletion mechanism. Another important observation is that GraphTinker's delete-only mechanism causes a throughput degradation as more edges are deleted from the database, whereas the delete-and-compact-in mechanism shows stable performance. This is because, with the delete-only mechanism, there is no shrinking of the data structure and so the same time has to be spent following edges, even though the number of edges in the database decreases. Whereas with the delete-and-compact mechanism, the data structure shrinks as more edges are deleted, allowing less time spent in following edges and more stable throughput.

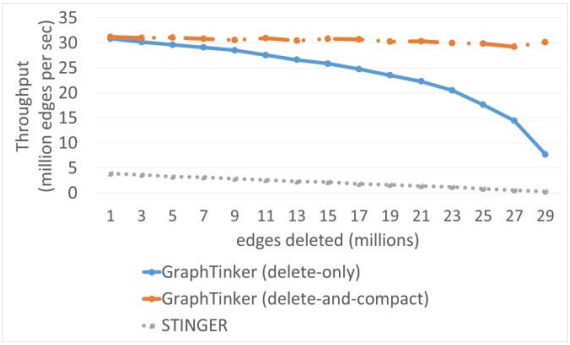

Fig. 15: Throughput for GraphTinker vs. STINGER when running BFS on the RMAT_2M_32M dataset and with different number of edges deleted.

Fig. 15 show the effect of the deletion mechanisms on analytics when both data structures (GraphTinker and STINGER) are used for graph analytics. The y-axis represents throughput (in millions edges/sec) of graph analytics, while the x-axis represents the amount of edges deleted (in millions of edges).

As shown, the delete-and-compact mechanism outperforms the delete-only mechanism about 1.2X when half of the edges are deleted, to as much as 4X when the last batch is deleted. Both mechanisms also surpass STINGER's deletion mechanism. Another important observation is that the delete-only mechanism experiences degradation in throughput (from 30 million edges per sec in first batch to 7 million edges per sec on last batch) while the delete-and-compact mechanism experiences stable performance. The reason is similar to what was described earlier: with the delete-only mechanism, the data structure does not experience any shrinking and so the time spent retrieving edges from the database remains the same even as more edges are deleted.

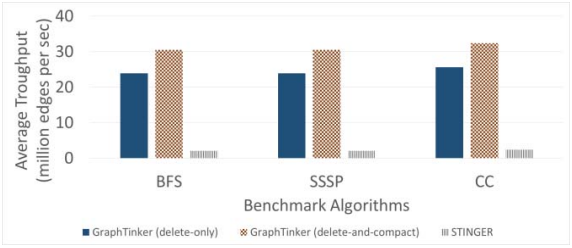

Fig. 16: Average processing throughput for GraphTinker vs. STINGER when running BFS, SSSP and CC algorithm on the RMAT_2M_32M dataset and performing edge deletions

Fig 16 shows the performances of BFS, SSSP, and CC algorithm running across the RMAT_2M_32M dataset. The y-axes in the figure represent the average throughput (in million edges/sec). The figure shows that the delete-and-compact mechanism demonstrates better performance, compared with the delete-only mechanism for all three algorithms.

Effect of different PAGEWIDTH sizes on GraphTinker's performance.

In order to evaluate the effect of different PAGEWIDTH sizes on GraphTinker's data structure and analytics performance, we configure GraphTinker on different PAGEWIDTH sizes (16, 32, 64, 128 and 256) and run the BFS algorithm on the Hollywood2009 dataset. This dataset was chosen arbitrarily.

Fig. 17 shows the performance of GraphTinker's data structure with the five different PAGEWIDTH sizes. The y-axis represents the insertion throughput while the x-axis represents the input sizes of edges loaded. Fig. 18, on the other hand, shows the corresponding BFS performance with the five different PAGEWDTH sizes when the graph engine was configured to run on incremental processing (IP) mode. This mode is selected because it utilizes the EdgeblockArray.

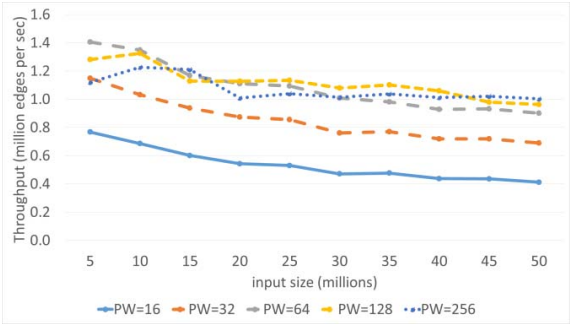

Fig. 17: Effect of different PAGEWIDTH sizes on insertion throughput to GraphTinker data structure when loading the Hollywood2009 dataset.

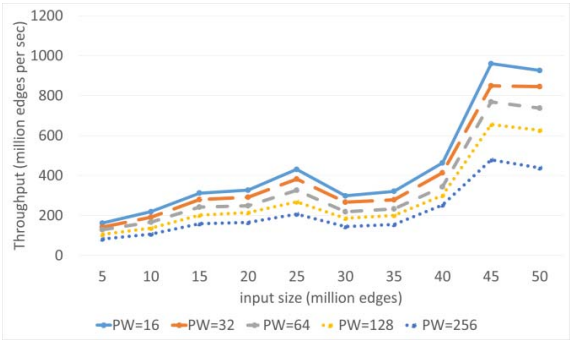

Fig. 18: Effect of different PAGEWIDTHs on graph analytics throughput when running the BFS algorithm on the Hollywood2009 dataset.

Fig 17 highlights two interesting behaviors. First, an increase in the PAGEWIDTH size from 16 to 256 demonstrates an increase in the insertion throughput experienced by the data structure. This is because increasing PAGEWIDTH size increases the hash range of edgeblocks in the data structure, leading to reduced frequency of collision experienced by the RHH algorithm. Second, increased PAGEWIDTH size allows the data structure to experience lesser degradation of throughput as more edges are inserted (i.e., better throughput stability), with PAGEWIDTH size of 256 experiencing the highest throughput stability.

On the other hand, Fig 18 shows that increasing the PAGEWIDTH decreases the throughput of graph analytics and vice versa. This is because smaller PAGEWIDTH sizes give a more compacted data structure compared to larger PAGEWIDTH sizes, and this allows more edges to be retrieved per unit time during graph analytics.

Choice of optimal PAGEWIDTH.

In order to find the most optimal PAGEWIDTH size for GraphTinker, we designed an experiment which investigates the performance with varying ratios of updates to analytics. This experiment projects two use cases of a dynamic graph: (1) where edge updates are made frequently and analytics rarely, and (2) where edge updates are made rarely and analytics frequently. The algorithm for analytics used for this experiment is the BFS algorithm. The ratios of updates to analytics in each of these experiments range from 1:10 to 10:1 and the PAGEWIDTHs used range from 8 to 256. For each dataset, 20 vertices among those with the highest degrees are pre-collected so that each analytic in each experiment uses a different root vertex. In each experiment comprising a dataset, PAGEWIDTH and updates/analytics ratio, edges are inserted into GraphTinker at batch sizes of 1 million edges per batch, and analytics are done in different intervals according to the update/analytics ratio of that experiment. The left part of the ratio (updates) determines how frequently the insertion process is intercepted in order to run graph analytics, while the right part of the ratio (analytics) determines how many analytics should be run on each interception. For example, with a ratio of 4:7 on the RMAT_2M_32M dataset which contains 32 million edges, the edge insertion process to the graph is intercepted 4 times (i.e., after every 6 batches are inserted) in order to run 7 different analytics in each interception, each on a different choice of root vertex. After each experiment is conducted (360 experiments in total), we then calculate the average time lapses of each experiment using a given dataset and a given PAGEWIDTH (36 different data points). The result is then plotted as shown in figure 19.

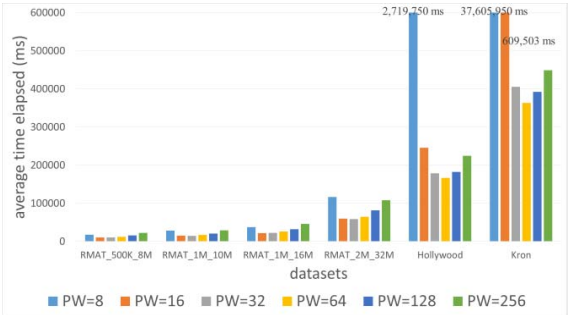

Fig. 19: Behaviors of different PAGEWIDTHs in a combination of updates and analytics using different datasets when running the BFS algorithm. Bars are averaged across updates/analytics ratios.

The y-axis of Fig 19 represents the time elapsed (in milliseconds) averaged across the updates/analytics ratios of each dataset and each PAGEWIDTH combination, while the x-axis represents the different datasets used in the experiments. As shown, the PAGEWIDTH size of 64 demonstrates the best overall performance, especially when dealing with larger datasets. Lower PAGEWIDTH sizes

such as 8 exhibit poor performance due to very low edge-update performance, which becomes more pronounced with larger datasets. Although higher PAGEWIDTH sizes exhibit good edge update performance, they ultimately experience poor analytics performance due to their less compacted arrangement of graph edges. The PAGEWIDTH size of 64 appears to represent a good balance of update/analytics ratio. These experiments were conducted with the BFS algorithm. Nevertheless, we expect this behavior to generalize to other GAS based algorithms using the edge centric model because the experiment illustrates the memory access behavior of the BFS algorithm, which is the same model as any other GAS based analytics algorithm.

VI. CONCLUSIONS

Our paper presents GraphTinker, a high-performance data structure for dynamic graph processing, as well as a new hybrid graph engine to improve efficiency when processing dynamic graphs. Technical advances offered by GraphTinker include: (1) a novel data structure that combines the benefits of two well-known hashing schemes to reduce the probe distance while following edges, and hence provides better solutions to the state-of-the-art data structure models based on adjacency lists, and (2) Scatter-Gather and Coarse Adjacency List (CAL) features that allow efficient compaction to our data structure to minimize DRAM accesses. The hybrid graph engine offers an improvement over the store-and-static-compute model and the incremental-compute model proposed in previous works. This is achieved by automatically selecting the best execution path between these two modes for every iteration in order to combine the advantages offered by both models.

Evaluation of GraphTinker on a variety of datasets and graph algorithms demonstrates that GraphTinker is capable of providing up to 3.3X performance improvement in update throughput on the CPU over the previous state of the art, STINGER. When used to run algorithms such as BFS, SSSP and CC on dynamic graphs, GraphTinker's more efficient data structure enables up to 10X improvement in performance compared to STINGER when running analytics in full processing (FP) mode. Finally, experimental tests using our hybrid graph engine demonstrate its capability to provide up to 2X performance improvement over the incremental processing model and up to 3X performance improvement over the full processing model.

ACKNOWLEDGEMENTS

This work was supported in part by CRISP, one of the six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by grant no. MF 17-027-IT from the Virginia Center of Innovative Technology. We would also like to thank the anonymous reviewers for their very helpful suggestions.

REFERENCES

[1] Sengupta, D., et al. "GraphIn: An Online High Performance Incremental Graph Processing Framework". European Conference on Parallel and Distributed Computing (Euro-Par), Aug., 2016.

[2] Murphy, R.C., Wheeler, K., Barrett, B., and Ang, J.A. "Introducing the Graph 500". Cray User's Group (CUG), Jan., 2010.

[3] Celis, P., Larson, P., and Munro, J., "Robin Hood Hashing". 26th Annual Symposium on Foundations of Computer Science (FOCS), pp 281-288. Oct., 1985.

[4] Mitzenmacher, M., "A New Approach to Analyzing Robin Hood Hashing". Proc. Thirtieth Workshop on Analytic Algorithmics and Combinatorics (ANALCO). Society for Industrial and Applied mathematics, pp 10-24. Jul., 2014.

[5] Devroye, L., Morin, P., and Viola, A., "On Worst-case Robin Hood Hashing". Society for Industrial and Applied Mathematics (SIAM) Journal on Computing, 33(4):923–936. May, 2004.

[6] Ediger, D., McColl, R., Riedy, J., and Bader, D., "Stinger: High performance Data Structure for Streaming Graphs". Proc. IEEE Conf. on High Performance Extreme Computing (HPEC), Sep., 2012.

[7] University of Florida Sparse Matrix Collection. http://tinyurl.com/hh8g3n9

[8] Sengupta, D., Song, S.L., et al., "GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems". Proc. ACM/IEEE Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC), Nov., 2015.

[9] Malewicz, G., Austern, M.H., Bik, A.J., et al., "Pregel: A System for Large-scale Graph Processing". Proc. SIGMOD Conf., on Management of Data June, 2010.

[10] Gonzalez, J. E., Low, Y., Gu, H., et al. "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs". Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI), Oct., 2012.

[11] Low, Y., Bickson, D., Gonzalez, J., et al., "Distributed graphlab: A Framework for Machine Learning and Data Mining in the Cloud". Proc. International Conference on Very Large Data Bases (VLDB), Apr., 2012.

[12] Han, W., Miao, Y., Li, K., et al., "Chronos: A Graph Engine for Temporal Graph Analysis". Proc. European Conf. on Computer Systems (EuroSys), Apr., 2014.

[13] Sun, J., Faloutsos, C., Papadimitriou, S., and Yu, P. S., "Graphscope: Parameter-free Mining of Large Time-evolving Graphs". Proc. ACM Int'l Conf. on Knowledge Discovery and Data Mining (KDD), Aug., 2007.

[14] Kyrola, A., Blelloch, G., and Guestrin, C., "Graphchi: Large-scale Graph Computation on Just a PC". Proc. USENIX Conf. on Operating Systems, Design and Implementation (OSDI), Oct., 2012.

[15] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-stream, Edge-centric Graph Processing Using Streaming Partitions". Proc. ACM Symp. on Operating Systems Principles (SOSP), Nov., 2013.

[16] Sengupta, D., Song, S.L., et al., "Graphreduce: Processing Large-Scale Graphs on Accelerator-based Systems". Proc. ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage, and Analysis (SC), Nov., 2015.

[17] Ediger, D., Jiang, K., Riedy, J., and Bader, D., "Massive Streaming Data Analytics: A Case Study with Clustering Coefficients". IEEE Int'l Symp. on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Apr., 2010.

[18] McColl, R., Green, O., and Bader, D., "A New Parallel Algorithm for Connected Components in Dynamic Graphs". Proc. IEEE Int'l Conf. on High Performance Computing (HiPC), Dec., 2013.

[19] Twitter statistics 11. Twitter Statistics. http://tinyurl.com/kcuhdcw

[20] Ramalingam, G., and Reps, T. "An Incremental Algorithm for a Generalization of the Shortest-Path Problem". Journal of Algorithms, 21(2):267-305., Sept., 1996

[21] Ham, T. J., Wu, L., Sundaram, N., Satish, and Martonosi, M., "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics". In Proc. of the ACM/IEEE International Symp. on Microarchitecture (MICRO), Oct., 2016.

[22] Fan. J., Raj A. G. S., and Patel. J. M., "A Case Against Specialized Graph Analytics Engines". In Biennial Conf. on Innovative Data Systems Research (CIDR), Jan., 2015.