Storing Streaming Key-Value Pairs in Aspen

Jayjeet Chakraborty and Nilesh Negi
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, CA, USA
jayjeetc@ucsc.edu, nnegi@ucsc.edu

***Abstract*—With the ever-increasing amount of data streams generated around the world, efficient processing and storage of streaming data is more important than ever. Streaming data is generally in the form of Graphs and Key-Value pairs representing everything from social network relationships to metadata. Present day stream processing engines use B-Tree like structures to ingest data, which although good for storage is inefficient for fast inserts, updates or deletes due to poor data locality and high space overheads. Also, there is no opportunity of compression in B-Trees. C-Trees or compressed purely-functional search trees stand as a solution to this problem by providing snapshot-based immutability semantics. It also stores data as chunks in its nodes enabling compression and savings in storage space. Aspen is a graph processing framework that uses C-Trees at its core to store graph data and perform batch as well as stream processing. We aim to utilize the core C-Tree functionality of Aspen and enable it to store streaming key-value pairs, in addition to Adjacency Graphs as key-value pairs form a large segment of all the streaming data generated around the world.**

This paper provides an initial benchmark on how C-Trees provide performance benefits over B-Trees in terms of integer-based key-value pair ingestion and deletion and set the context of being able to use C-Trees to store key-value pairs with complex value types such as string, binary, and maps.

***Index Terms*—stream processing, compressed purely-functional search trees, aspen, key-value pairs, aspen, ligra**

I. Introduction

As compute and connectivity became cheaper over the past two decades, there has been an exponential increase in the amount of data produced around the world. Most of this data is streaming data in the form of Graphs and key-value pairs [6]. Graph data mostly represents relations between different entities while key-value data represents metadata or even actual data. This immense growth in streaming data can be attributed to the increase in the number of Edge and IoT devices such as smart home appliances, sensors, logging applications, real-time monitoring systems, and even mobile applications which continuously emit streams of data. Such kind of streaming data is important for performing analytics in real-time, making import business decisions fast, and even finding out security vulnerabilities in large-scale systems.

The most common types of operations on streaming data is inserts, updates, and deletes. When streams of data reach the stream processing systems, they are ingested into the system, persisted in an object storage system such as S3 and GCS [3], and analytics functions and queries are run on them concurrently. Processing of streaming data is mostly done in windows using a sliding-window like technique. The stream processing engine is usually connected to a Dashboarding system such as Katana or Graphana where the analysis results are aggregated and updated in real-time. At the core of the data ingestion engine of most stream processing systems is a B-Tree or a B+Tree [4] like structure which stores the key-value or vertex-vertex pairs (sometimes ordered by their key or vertex) for logarithmic time inserts and searches.

The issue with using B-Tree like structures in the ingestion or storage engine of stream processing systems is that when data is updated, the nodes of the B-Tree needs to be mutated. Since, the tree is mutated every time, operations reaching the tree such as inserts, updates, and deletes need to be serialized and hence no parallelism can be incorporated inherently. Additionally, each node of the B-Tree stores just a single data point disabling any opportunity for compressing the data stored.

To fix such issues with B-Trees, Dhulipala et. al. came up with Compressed Purely-Functional Search Trees in 2019 [1]. We refer to this tree as C-Tree in the rest of the paper. C-Tree's are immutable, which means when an insert or update operation executes on the tree,

the targeted nodes create a copy of themselves as the new version of the node and point to the old version of itself. Thus a copy-on-write kind of semantics is followed. Such an approach keeps the tree immutable, so when multiple operations try to update the same set of nodes in the tree, they can be done in-parallel on the different versions and later aggregated. The outcome of this is parallelism can be used while doing operations on the tree. Also, C-Trees stores several data points together in a vector at each node, which allows doing batch operations on the tree and most importantly allows compressing the data stored in each node. This reduces the overall disk or memory usage while storing the tree.

The Aspen system uses C-Trees to ingest and store streaming Graphs efficiently. This works fine, but the problem was that Aspen can only work with Graphs and not generic key-value pairs which are also a substantial segment of all the streaming data generated in the world. The goal of this project was to modify Aspen so that it can store simple key-value pairs apart from Graphs. We implement a benchmark for inserting and deleting [int,int] key-value pairs in Aspen's implementation of C-Tree and in a simple B-Tree. Our results show that writing data to C-Trees are an order of magnitude more efficient than doing so in B-Trees. The contributions of the paper are listed as follows:

- A prototype code to check if we can simply store [int,int] pairs in a C-Tree without using a Graph interface.
- A performance evaluation to find out the performance advantages of doing inserts and deletes in C-Trees over B-Trees.
- Ongoing work to make Aspen support string/binary data types as values, so that [int,string] pairs can also be ingested and compressed in Aspen.

This paper is organized as follows: Section 2 provides a background on the different concepts and technologies used in this project, Section 3 discusses the motivating factors behind this project, Section 4 discusses the all limitations that past work had, Section 5 gives an in-depth explanation of our contributions, then in Section 6, we provide an experimental evaluation of our implementation and discuss our results, and finally in Section 7 and 8, we discuss our future plans for this project and present our concluding thoughts.

II. Background

In this section, we discuss some background information on the technologies and concepts related to the work in this paper.

A. Batch Processing Systems

Batch Processing [13] refers to the practice of processing volumes of data in batches automatically. This is also referred to as the "offline processing" of data [9]. While users are required to submit the jobs, no other interaction by the user is required to process the batch. Batches may automatically be run at scheduled times as well as being run contingent on the availability of computer resources. Batch processing is an incredibly cost effective way to process huge amounts of data in a small amount of time. Because batch processing is largely automated, it does not require manual intervention. The automation reduces operational costs and increases the speed at which transactions and data can be processed. Bulk database updates, automatic extract-transform-load (ETL) in data warehouses, and performing bulk operations on digital images such as resizing, conversion, and watermarking are examples of batch processing. Apache Hadoop based on the Map Reduce paradigm and Apache Spark can be considered as popular batch processing systems. An example batch processing application is shown in Figure 1.

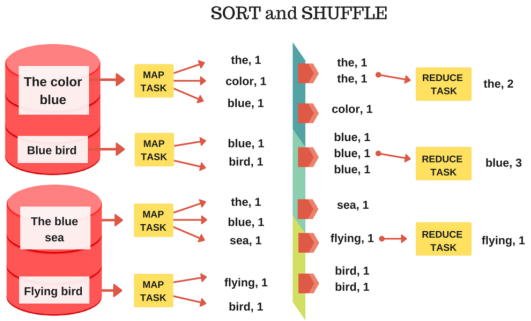

Fig. 1. "Sort and Shuffle" as an example batch processing application in Map Reduce

B. Stream Processing Systems

Stream Processing refers to the practice of running a processor on a window of streaming data as soon as they arrive to continuously generate insights and analytics. This is also referred to as the "real-time processing" of data. Stream processing also sometimes works in the pub-sub (publisher-subscriber) model where a publisher continuously publishes data to a stream processing engine, where the data is stored in an object-storage layer, analyzed, filtered, and processed, and then finally, a consumer subscribed to a specific topic to read this data in windows. The sources of data in stream processing systems are mostly IoT sensors, payment processing

systems, and server and application logs. Also, some of the common use cases of stream processing include real-time fraud and anomaly detection, IoT and edge analytics, and real-time personalisation, marketing, and advertising. Figure 2 shows the general architecture of Stream processing systems. Some popular stream processing systems in the market include Apache Storm, Apache Spark Streaming, Apache Kafka, Rising Wave, and Red Panda [6].

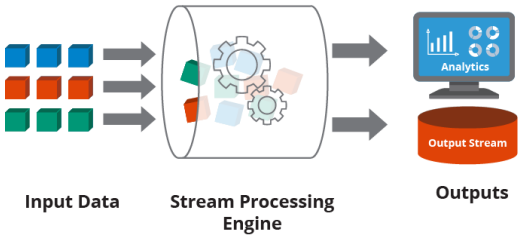

Fig. 2. Stream Processing Systems Architecture

C. Graph Processing Systems

A major portion of all the data produced in the world is in the form of Graphs. Graph data represented relationships in social networks, dependencies in a package manager, routers in a network, etc. These are mostly property graphs where the vertices have a unique ID and attached properties to it while edges consist of two unique IDs representing vertices on each side and an associated property field. There are several abstractions for Graph processing, the most popular being the Pregel abstraction. In the Pregel abstraction, individual vertex programs run in parallel on different vertices and use message passing to coordinate amongst themselves. Finally, the program stops when all the vertices vote to stop. While other Graph Processing Systems follow what is called the GAS paradigm, Gather-Apply-Scatter. Streaming Graph processing systems are those systems that process streaming graphs. They are of two types: static and dynamic. In static streaming graph processing systems, only new nodes are added to the graph, while in dynamic streaming graph processing systems, apart from new nodes being added, the values in the old graph nodes are also updated. Some popular Graph processing systems out there include GraphLab, Apache Giraph, Google Pregel, and Apache Spark GraphX [11]. An example Graph processing pipeline is shown in Figure 3.

D. Purely-Functional Search Trees

Generally, a data structure is called a purely-functional data structure [8] when it can be implemented in

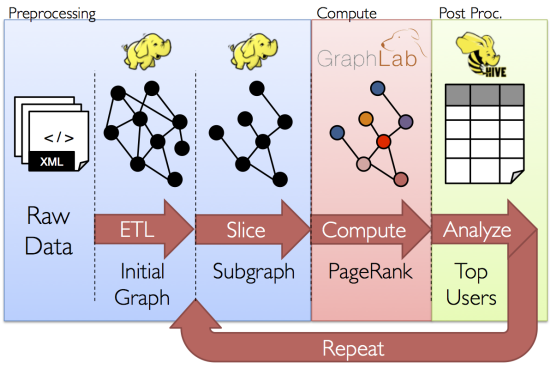

Fig. 3. Example Graph Processing Pipeline

a purely-functional programming language such as Haskell. Another way to look at it is that purely functional data structures are immutable in nature. By immutable we mean that these data structures when modified keep an unmodified version of themselves intact and use pointers to make the new version point to the older versions. Binary trees when implemented in this way make purely functional trees. Inherent immutability in purely functional search trees provide the ability to have parallelism and thus improved throughput and reduced latency while doing search tree operations. Apart from all its benefits, Purely functional search trees have some downsides such as increased space usage (as it needs to maintain multiple versions of the same node) and hence poor data locality.

E. Compressed Purely-Functional Search Trees

A C-Tree (Compressed Purely Functional Tree) extends the functionality of Purely Functional Trees with chunking. It is a kind of binary tree with chunking at every node, following a chunking scheme that allows storing multiple elements per node contiguously in an array for better data locality. As discussed in Dhulipala et al. [1], the chunking scheme in a C-Tree takes an ordered set of elements to be stored in the tree and promotes certain elements to become the head node of the tree. The remaining elements are stored in tails associated with each node. C-Trees ensure that the same keys are promoted in different trees by using a hash function to choose which elements need to be promoted as head nodes.

C-Trees differ from B-Trees in the fact that B-Trees are more focused on keeping the height of the tree minimum so that searchers are faster, whereas in C-Trees the goal is to store many contiguous segments in a single node to achieve better compression. Another problem with B-Trees is that in a purely functional setting,

copying the nodes in a path (path copying) is necessary for updates. But in C-Trees, updates can be achieved by only copying a single node and making it point to the old node. Also, not updating the original nodes makes C-Trees immutable in nature that allows multiple parts of the tree to be worked upon in parallel. This improves runtime when performing updates in multiple parts of a C-Tree because these different updates can be done concurrently. Thus, the main benefit of using C-Trees over B-Trees is reduced space usage due to unlocked compressibility and faster batch inserts and deletes. A Purely-Functional C-Tree structure is shown in Figure 4.

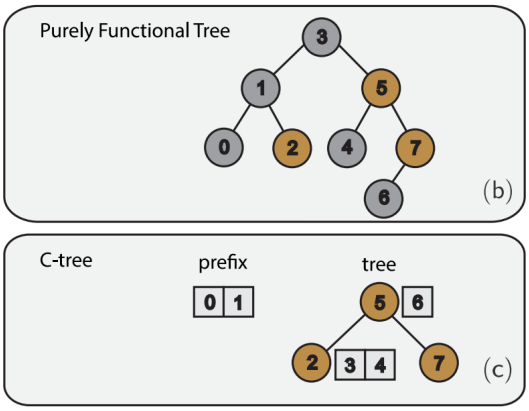

Fig. 4. Structure of a Purely-Functional C-Tree

F. Aspen: Low-Latency Graph Streaming System

Aspen introduced the C-Tree data structure as described earlier and uses this as the central idea. In addition to chunking for better data locality, Aspen assumes that each chunk in the C-Tree stores a sorted array of integers and uses Difference Encoding within each chunk for compression. Difference Encoding can be visualized as: "Given a chunk containing d integers $\{I_1, I_2, ..., I_d\}$, differences can be computed as $\{I_1, I_2 - I_1, ..., I_d - I_1\}$

Aspen is implemented in C++. It uses a custom datatype for representing `[int, int]` inputs and has custom subroutines for building the C-Tree, inserting nodes, deleting nodes, and conducting queries on the C-Tree. The input Adjacency Graph which simulates an input stream of integer values is used to generate a treeplus_graph as part of `graph/api.h`.

Additionally, Aspen implements a snapshot functionality in `immutable_graph_tree_plus.h`to maintain graph immutability between different update operations. This snapshot subroutine can use uncompressed (`uncompressed_nodes.h`, `uncompressed_iter.h`, `uncompressed_lists.h`) or compressed (`compressed_nodes.h`, `compressed_iter.h`, `compressed_lists.h`) subroutines. These subroutines together help form the tree_plus data structure in `tree_plus.h`

III. Motivation

The Aspen Streaming Graph Processing framework was designed to store graphs efficiently using less space and with better data locality. This enables fast batch updates, inserts, and deletes. Graph data inputs consist of [int, int] pairs representing two connected vertices via a tuple of offsets and edges. This is semantically similar to simple key-value pairs, although the value in key-value pairs extends to include string, binary, or any other complex/nested data type. Data pairs in Graphs and Key-Value streaming systems are depicted in Figure 5. So, the idea is to use the benefits of C-Tree's and Aspen for simple key-value stream inputs, updates, and deletes, as from a high level both the inputs are indistinguishable apart from the data types each workload needs to support. Many popular key-value stores use a B-Tree or a B+Tree-like data structure to store their data. The goal is to check how replacing B-Trees with C-Trees results in space and runtime improvement in key-value processing. In the next section, we discuss the limitations of Aspen that currently prevent it from processing streaming key-value pairs.

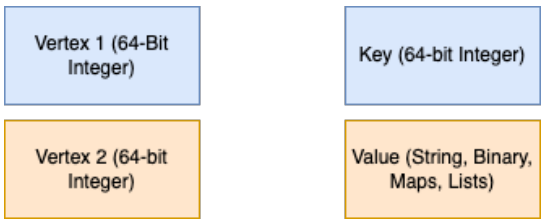

Fig. 5. Streaming Graph Input vs Streaming Key/Value Input

IV. Limitations of Aspen for K/V Workloads

The Aspen framework is designed to only ingest and process graph data containing vertices and edges. Additionally, Aspen expects its input to be in a particular "Adjacency Graph" format from the Problem Based Benchmark Suite [5]. In this format, the inputs start off as a sequence of offsets, one for each vertex, followed by a sequence of directed edges, ordered by their source vertex. The offset for a vertex i refers to the location of the start of a contiguous block of out edges for vertex i in the sequence of edges. The block continues until the

offset of the next vertex, or the end if i is the last vertex. All vertices and offsets are 0-based and represented in decimal. This format is shown in Figure 6.

```
AdjacencyGraph
<n>
<m>
<o0>
<o1>
...
<o(n-1)>
<e0>
<e1>
...
<e(m-1)>
```

Fig. 6. Adjacency Graph format of Aspen (Ligra)

Such an input format is not practical for storing real-world key-value pairs, where the values are mostly strings/binaries or maybe even more complex data types. Apart from this, the dataset input format is hard-coded in Aspen to expect'Adjacency Graph'. Additionally, as the source code of Aspen is based on Ligra [12], it is hard-coded to use long/long-long integers only as vertex pairs (representing edges). So, after browsing the source code, to be able to make Aspen store and process streaming key-value pairs (int-string/binary), the input format, the input processor, C-Tree, C-Tree APIs, and everything related needs to be refactored. In the next section, we discuss the details of our implementation.

V. Implementation

We base our implementation on top of Aspen, the streaming graph processing framework on top of the Ligra interface. Aspen has multiple interfaces inside it all nested using C++ templates. At the core is a 'tree_plus' interface defined in `code/graph/tree_plus/tree_plus.h`, which is an implementation of a purely-functional tree. 'tree_plus' has 2 variations: one with uncompressed nodes and one with compressed nodes. These abstractions are defined in `code/graph/tree_plus/compressed_nodes.h` and `code/graph/tree_plus/uncompressed_nodes.h`. Finally, at the top level, we have a 'versioned_graph' interface, which defines a graph represented using C-Tree. This interface is defined in `code/graph/versioned_graph.h`.

Since Aspen already used a C-Tree internally for storing Graphs, we started by extending the C-Tree interface for storing key-value pairs. Originally, a C-Tree stores vertex-vertex pairs representing the edges of a Graph. We can observe this is `code/common/types.h` and `code/common/IO.h`. So, we started by implementing new data types for Keys and Values, where Keys are represented with integers (or) long integers, and Values are represented with C++ character arrays. The choice of using integers or long integers for Keys can be done via a compilation flag `-DKEYLONG`.

```
// Defining data types for Key and Value

// define KEYLONG
#if defined(KEYLONG)
typedef long intK;
typedef unsigned long uintK;
#define INT_K_MAX LONG_MAX
#define UINT_K_MAX ULONG_MAX
#else
typedef int intK;
typedef unsigned int uintK;
#define INT_K_MAX INT_MAX
#define UINT_K_MAX UINT_MAX
#endif

// String Values
typedef char* strV;
```

Fig. 7. Modified code/common/types.h

After adding support for Key-Value pairs, we modified Aspen's file reading subroutine `read_unweighted_graph` in `code/graph/IO.h` to accept input files that do not adhere to the 'Adjacency Graph' format. As per this format:

- The first line in the input file is a string "Adjacency Graph".
- The second line is an integer representing the count for the number of offsets.
- The third line is an integer representing the count for the number of edges.
- After these 3 lines, all offsets appear in newlines followed by all edges.

The default tokenizer for this 'Adjacency Graph' format splits the input stream into tokens by using any whitespace as a delimiter. For Key-Value Pairs, we created a new subroutine `read_unweighted_kv_graph` in `code/graph/IO.h` and a new tokenizer `pbbs::sequence Lines` in `code/pbbslib/strings/string_basics.h`.

```
template <class Seq, class UnaryPred>
sequence<char*> lines(Seq &S, UnaryPred const &is_newline) {
  size_t n = S.size();
  timer t("lines",false);

  // clear newlines
  parallel_for (0, n, [&] (size_t i) {
if (is_newline(S[i])) S[i] = 0;}, 10000);
  S[n] = 0;
  t.next("clear");

  auto StartFlags = delayed_seq<bool>(n, [&] (long i) {
return (i==0) ? S[i] : S[i] && !S[i-1];});

  //Character count length
  auto Pointers = delayed_seq<char*>(n, [&] (long i) {
return S.begin() + i;});
```

Fig. 8. Modified Tokenizer

This can accept Key-Value inputs of type `[int, char*]` where the Value can have any alphanumeric characters or symbols such as:

```
1020253529628262401 # #YouTube # # # # #
1020253529544577024 #FormPro
1020253529309528065 #TheGreatSeungri #KINGSEUNGRI_123
1020253528735010817 #radicalismos #Libertad #Democracia #Paz #Europa #Mundo #StopRadicalismos
1020253528680554496 #LateNightWhenever
1020253528638533632 #FridayFreebie
1020253528558690306 #phillyhereicome
1020253528453832708 # #
1020253528365953025 #projeto60kg #foconameta #instafit #fitnessgirl #agoraeuficotop
```

Fig. 9. ID-Hashtag dataset

```
1020235969885134848 (-115.223125,36.232915)
1020235968429920258 (-83.804475,27.698681999999998)
1020235971546185728 (-80.16260199999999,25.6969375)
1020235971671932928 (-115.133373,36.249124999999999)
1020235971839881216 (-84.433106,33.7671945)
1020235971160260608 (-117.1911905,34.257528)
1020235972150284289 (-110.0447835,46.6798)
1020235973907673093 (-87.8211335,41.811298)
1020235974943617024 (-86.6807375,32.576227)
```

Fig. 10. ID-Geo dataset

The aforementioned figures 9 and 10 represent real-world datasets from a public source [14].

After modifying the file ingestion subroutines, we started adding subroutines that govern the generation of the C-Tree using Key-Value inputs. To simplify our starting effort, we focused on uncompressed graphs and used subroutines like `initialize_treeplus_kv_graph` and `initialize_kv_graph` in `graph/api.h`. This led to modifying the template definitions for `traversable_graph<sym_immutable_graph_tree_plus>` in `code/graph/versioned_graph.h`, `code/graph/traversible_graph.h`, and `code/graph/tree_plus/immutable_graph_tree_plus.h`. However, due to lack of time, we were unsuccessful in completing the code porting in these sections to support key-value based input streams.

Since we were unsuccessful in supporting key-value inputs end-to-end in Aspen, we devised a proxy benchmark that can help measure the effectiveness of C-Trees over B-Trees for Key-Value inputs. We figure that vertex-vertex pairs actually mimic a key-value pair where both the key and value are 64-bit integers. The C-Tree data structure is implemented in the `immutable_graph_tree_plus.h` class in Aspen. It has a bunch of APIs to insert, delete, and query the contents of a C-Tree. The most important of these APIs are the 'insert_edges_batch' and the 'delete_edges_batch' APIs that are used to insert and delete a batch of edges (Key/Value pairs) in Aspen. We write a C++ code where we use the PBBS (Problem Based Benchmark Suite) library to generate random key-value pairs, insert them into an empty C-Tree, and then delete them. We do the inserts and deletes in two ways. One, we insert/delete the key-value pairs one-at-a-time and second, we do the same in batches. We measure the time required to insert and delete $1,000,000$ key-value pairs to/from a C-Tree for our performance evaluations.

VI. Evaluations

We evaluate the performance benefit of inserting key-value pairs in C-Trees by comparing its performance to B-Trees. For the B-Tree implementation, we create a simple B-Tree implementation that stores key-value pairs in its nodes. We use hardware from CloudLab [10], an NSF-funded bare-metal as a service for our experiments. We use the machine codenamed 'm510' consisting of 8-core Intel Xeon D-1548 at 2.0GHz, 64GB ECC Memory, and 256GB NVMe SSD from CloudLab.

As described in the previous section, we used randomly generated key-value pairs as our dataset. From our experiments, we found that the time to insert and delete key-value pairs in a C-Tree is significantly better than doing the same in a B-Tree, about an order of magnitude. Our results are shown in Figure 11.

We believe that the significant performance difference that we observe is due to the fact that C-Trees do not create copies on inserts rather they create a new node and point to the old node. The original tree is never mutated and hence an approach of immutability is followed. Using such an immutable data structure, the C-Tree, inserts, and updates can be done in parallel which the 'insert_edges_batch' method utilizes. This unlocked parallelism gives us the performance benefits that we observe.

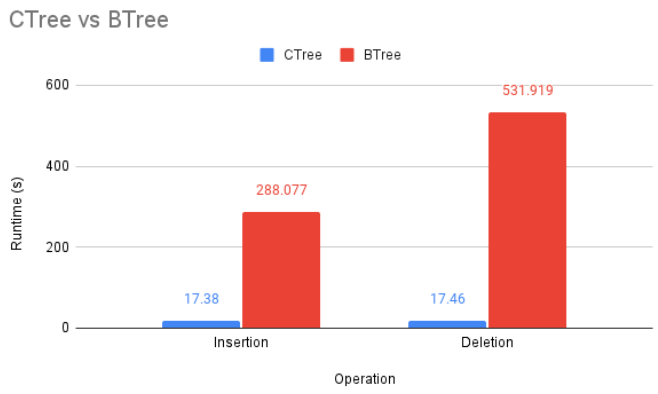

Fig. 11. Performance comparison of insertion and deletion of key-value pairs in C-Tree versus B-Tree

VII. Future Work

As future work, our main goal is to complete the code port for supporting key-value pairs in Aspen where the value can be any arbitrary data type, starting with a string/binary data type. For this, we need to complete the code port in the Versioned Graph interfaces and the Tree Plus interfaces so that they support string data types as values. We believe that the best way to achieve this is by adding adjacent interfaces to the pre-existing ones that basically support `[int, string]` pairs. Also, once these interfaces are implemented and we ensure that it is working, we need to add compression support for the C-Tree leaves. We plan to add support for compression algorithms such as Snappy, ZSTD, and LZ4 [7]. Finally, we plan to write unit tests for all the functionality we added and conduct performance benchmarks to evaluate our implementation.

VIII. Conclusion

Data is being produced continuously in today's world. Most data producers work round the clock and keep producing streams of data mostly in the form of Graphs. Streaming Graph data is important for gaining real-time insights and taking decisions fast. There are several stream processing systems out there for storing and processing data streams such as Apache Storm and Apache Spark Streaming. These systems work fine for inserts, but they mostly perform poorly during updates. This is mainly because, during updates, the original node in a B-Tree or a B+Tree is updated, and this mutation is expensive. Also, since the original tree is mutated, this operation can't be parallelized. Compressed Purely Functional Trees (C-Trees) come into the picture as a replacement for B-Trees to solve this exact problem. These are a class of immutable search trees, where when data is inserted or any node is updated, a new node is created with the new data and made to point to the old node. This way there's no need for mutation and hence multiple of these insert and update operations can be done in parallel. Therefore, these trees have batch insert, update, and delete APIs where inserts, updates, and deletes of individual key-value pairs are done in parallel. In this work, we aim to expand the scope of C-Trees from storing only Graphs (vertex pairs representing edges) to storing simple key-value pairs (pairs of int and strings) which represent a wider range of streaming workloads in the real world. We use the batch insert and delete APIs of the C-Tree interface in Aspen and store simple key-value pairs in it. For now, we only insert integer-integer pairs, but we aim to modify the source code enough so that we can store more complex value types such as string, binaries, maps, etc. We measure the performance difference in inserting and deleting key-value pairs from both a C-Tree and a B-Tree and found that insertion and deletion operations are about an order of magnitude faster in C-Trees than in B-Trees. We attribute this performance gain to the parallelism that C-Trees provide due to their immutability.

Acknowledgment

This project has been done as a part of the CSE 293 - Advanced Topics in Computer Systems class offered during Winter 2023 at the University of California, Santa Cruz. We would like to thank Professor Liting Hu and her TA Yinzhe Zhang for their immense help and guidance throughout this course.

References

[1] Dhulipala, Laxman, Guy E. Blelloch, and Julian Shun. "Low-latency graph streaming using compressed purely-functional trees." Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation. 2019.

[2] Fragkoulis, M., Carbone, P., Kalavri, V. and Katsifodimos, A., 2020. A survey on the evolution of stream processing systems. arXiv preprint arXiv:2008.00842.

[3] Factor, M., Meth, K., Naor, D., Rodeh, O. and Satran, J., 2005, June. Object storage: The future building block for storage systems. In 2005 IEEE International Symposium on Mass Storage Systems and Technology (pp. 119-123). IEEE.

[4] Comer, D., 1979. Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2), pp.121-137.

[5] Problem Based Benchmark Suite (2020) http://www.cs.cmu.edu/ pbbs/benchmarks/graphIO.html

[6] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J. and Poulosky, P., 2016, May. Benchmarking streaming computation engines: Storm, flink and spark streaming. In 2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW) (pp. 1789-1792). IEEE.

[7] Alakuijala, J., Kliuchnikov, E., Szabadka, Z. and Vandevenne, L., 2015. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. Google Inc, pp.1-6.

[8] Wikipedia contributors, Purely functional data structure — Wikipedia, The Free Encyclopedia, 2022,
[9] Wikipedia contributors, Batch processing — Wikipedia, The Free Encyclopedia, 2022,
[10] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K. and Akella, A., 2019, July. The Design and Operation of CloudLab. In USENIX Annual Technical Conference (pp. 1-14).
[11] Batarfi, O., Shawi, R.E., Fayoumi, A.G., Nouri, R., Beheshti, S.M.R., Barnawi, A. and Sakr, S., 2015. Large scale graph processing systems: survey and an experimental evaluation. Cluster Computing, 18, pp.1189-1213.
[12] Shun, J. and Blelloch, G.E., 2013, February. Ligra: a lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (pp. 135-146).
[13] Shahrivari, S., 2014. Beyond batch processing: towards real-time and streaming big data. Computers, 3(4), pp.117-129.
[14] Hyuk-Yoon Kwon, Real and synthetic data sets for benchmarking key-value stores focusing on various data types and sizes, Data in Brief, Volume 30, 2020, 105441, ISSN 2352-3409, https://doi.org/10.1016/j.dib.2020.105441. (https://www.sciencedirect.com/science/article/pii/S2352340920303358).