Changes to the Manuscript

The authors thank the reviewers for their comments. In this updated manuscript, we focused on clarifications.

Reviewer n3LE raised questions about the motivation behind SnapKV. Our approach is designed with the recognition that current OLDA systems separate the concerns of computation/analytics from the concerns of transactional processing. This can lead to operations occurring on stale data. Recent work in HTAP, however, recognizes the need to unify these systems rather than utilizing custom middleware. It simplifies reasoning about atomicity and isolation as well as minimizes staleness. Furthermore, recent work in HTAP, such as Vegito, recognizes the ability for more modern hardware architecture, in Vegito's case, offloading to RDMA, to support this approach. Our evaluation in comparison to both OLDA and HTAP solutions demonstrates that designing data repositories with heterogeneous workloads in mind, as well as the addition of GPUs as a coprocessor in these workloads, can achieve desirable performance and properties (freshness). The primary goal of this work is to design a system that achieves both freshness and performance, and can be utilized in both workloads.

Reviewers n3LE and u7h1 both raised questions about the high utilization and parallelism between the GPU and CPU. SnapKV utilizes a CUDA stream per transaction, which allows the system to have concurrent transactions also concurrently utilize the GPU. FeDB is also able to utilize multiple streams for computation. This enables the high GPU utilization. Even with aborts minimizing the amount of goodput SnapKV can achieve, it is still able to effectively move data between the devices at a rate that compensates for the extra complexity of using a GPU within a transaction. Since FeDB separates the repository and computation, we cannot move data between the database and the GPU as efficiently. Relaxing the transactional guarantee to achieve higher performance while minimizing the cost of data movement would be an interesting aspect to explore in the future.

In response to review u7h1, we added clarifications about the microbenchmark. The aim is to stress the need to move data between the CPU and GPU for computation. We do not make assumptions about the accuracy of the underlying model and instead intend to highlight the complex data movement that systems like FeDB employ.

Compared to Vegito, SnapKV does not employ a column store, which Vegito does. The key-value approach is less efficient for queries. We clarify this in the paper. We also stick to snapshot isolation, which achieves 0ms freshness, and therefore cannot make any tradeoffs with freshness and performance. It would be an interesting aspect to explore in the future.

In response to reviewer s9ri, we clarify the optimization opportunities and implementation details.

We also address all other minor comments from all reviewers.

SnapKV: Heterogeneous Key-Value Store for Heterogeneous Workloads

Anonymous authors
Paper under double-blind review

Abstract

Modern businesses increasingly require fast and accurate real-time decisions over concurrently modified data. The existing literature lacks a unified data repository supporting heterogeneous workloads (transactional, analytics, and machine learning inference). Prior work made significant trade-offs between freshness, performance, consistency, and generality. In this paper, we design SnapKV, a key-value store that supports heterogeneous workloads without sacrificing real-time order.

1 Introduction

Real-time analysis of data stored in transactional data repositories is becoming increasingly important. Databases like 4Paradigm's FeDB/OpenMLDB [5,13], PingCap's TiDB [24], Google's AlloyDB [20], Google's F1 Lightning [44], and Vegito [35] enable users to transact with databases and analyze the data concurrently. The workloads these systems process include hybrid transactional/analytical processing (HTAP) workloads and online decision augmentation (OLDA) workloads.

HTAP enables a database management system to perform both online transactional processing (OLTP) and online analytical processing (OLAP). Supporting these heterogeneous operations, which are substantially different in terms of computing demand, could lead to performance inferior to the performance of both OLTP-only or OLAP-only systems. This is mainly because accessing data while it is getting modified introduces bottlenecks within the concurrency control due to conflicts between long-running OLAP queries and short running OLTP transactions.

To address this bottleneck, recent solutions in literature have dealt with this issue by relaxing real-time guarantees and allowing for some staleness of queries in exchange for higher performance of transactional and analytical computation [24, 35]. However, the cost of staleness in high-performance HTAP databases can be too high for some applications like credit card fraud [13] (see scenario below). Ensuring data freshness in such applications, which is informally defined as the time between the commitment of a database transaction and when its modification can be observed by other transactions, is essential to reach decisions based on up-to-date data snapshots.

The need for more stringent freshness prompted the creation of a new computing paradigm named Online Decision Augmentation (OLDA) [13]. OLDA is a specific paradigm that allows for fast feature engineering to be used in machine learning models to improve the performance of the inference task. OLDA focuses on the specific kinds of analytics related to machine learning and allows for interfacing with machine learning inference services. Although existing OLDA systems indeed achieve high performance and high data freshness (i.e., real-time order), unfortunately they relax the transactional model in order to avoid the above bottlenecks. For example, FeDB/OpenMLDB [5,13], an OLDA database, does not allow for the use of the `BEGIN` and `COMMIT` SQL commands needed to write arbitrary transactions. As a result, performance increases, but programmability is severely affected.

The following example highlights an emerging need to maintain both freshness (through real-time order) and programmability (through general-purpose transactions that include analytics and machine learning inference computation) in modern data management systems. The example is inspired by credit card fraud detection. In a recent work [13], 4Paradigm [6], a company for banking fraud detection services, suggests that as people use credit cards, complex information stemming from the credit card transactions and their time relative to other transactions can aid in predicting fraud. In the same work, it is shown that fraud detection using machine learning inference primitives must occur within 20ms of the moment the transaction arrives.

Best effort approaches like those seen in recent HTAP databases [24, 35] do not guarantee such freshness and low latency. On the other hand, transactional support and real-time order are necessary properties to commit a financial transaction only if the most recent snapshot of execution is found to be not fraudulent.

Programmability through expressive APIs that allow for transactional operations, atomically executed along with non-conventional computation, such as machine learning inference, is essential for engineers to work with data repositories and analyze real-time data.

Motivated by the above observations, in this paper we introduce SnapKV, a heterogeneous data repository architecture that aims at achieving both programmability and freshness by (1) enabling transactional support along with real-time analytics, and (2) providing a general architecture to achieve high-performance transactions, analytics, and machine learn-

ing inference, all in an integrated infrastructure. The key to enabling SnapKV's heterogeneity is a new primitive called SNAPSHOT. This SNAPSHOT primitive builds off of prior work on linearizable range-queries [30, 43]. It allows programmers to perform a collection of range query operations that creates an arbitrarily large snapshot of data to be computed by OLDA and OLAP workloads. SNAPSHOT can be invoked within a transactional context, and SnapKV's concurrency control ensures that the snapshot remains consistent until the transaction commits.

In order to achieve all the aforementioned goals without sacrificing scalability, SnapKV follows the recent trend of integrating heterogeneous architectures, mainly graphics processing units (GPUs), in data repositories [17, 38, 45].

SnapKV's approach to heterogeneous computing is to divide transactions into smaller computational tasks and evaluate which architecture (i.e., CPU or GPU) may have more affinity to each task. We then evaluate the affinity of each task and design optimal functions/kernels for that architecture. More specifically to SnapKV, transactional primitives and data structures are more amenable to the CPU for these problems. However, when performing machine learning inference, we offload to the GPU. This type of approach to heterogeneity enables users to meet OLTP, OLAP, OLDA, and similar workload demands by utilizing high performance heterogeneous computation within their transactions.

We implemented SnapKV in C++ as a standalone system and tested it using enterprise CPUs and an NVIDIA Tesla V100 GPU. We compare its performance against FeDB [5, 13] for OLDA workloads, and against Vegito [35] and TiDB [24] for HTAP workloads. The results show that SnapKV achieves up to 2.43x speedup and 45.7% lower latency compared to FeDB. In HTAP workloads, SnapKV significantly outperforms TiDB, achieving up to 604x and 4x speedup in OLTP and OLAP throughput, respectively. Compared to Vegito, SnapKV's transactional throughput is improved by 1.3x, which matches our goals of providing real-time order guarantees without sacrificing transactional performance. On the other hand, SnapKV's OLAP throughput is 16.8% of Vegito's OLAP throughput. This is not surprising due to the highly optimized column-store design Vegito adopts, which is not the focus of this paper. Our future plans include investigating the possible extensions to include such optimizations in SnapKV.

SnapKV is released as open source project and has a publicly available artifact at: https://zenodo.org/records/13858237.

2 System Overview

SnapKV is an in-memory data repository designed to concurrently handle a variety of complex workloads including OLAP, OLTP, and OLDA (machine learning). SnapKV achieves this by providing two interoperating subsystems, the *transactional* and *compute* subsystems. Our transactional subsystem implements the transactional primitives, concurrency control, and in-memory storage. Support for OLAP operations and GPUs are enabled in the compute subsystem. More in detail:

- The transactional subsystem provides transactional primitives, including SNAPSHOT, which are all able to support the reads and modifications needed for each workload.
- The compute subsystem is able to provide relational algebra operations (e.g. join or sort) along with computational kernels (e.g. general matrix-matrix multiplications needed for deep learning inference) and GPU support in order to achieve the low latency required for these workloads. The compute subsystem supports dynamic loading so that custom operations and kernels can be added to enable future workloads to achieve their performance objectives.

These two subsystems are then utilized within a transaction, allowing programmers to consistently interact with data while optimizing analysis through CPU and GPU computation. Unlike prior approaches to HTAP or OLDA, SnapKV differs in how it fuses these two subsystems into a single centralized system.

SnapKV relies on the key-value data model, which provides a useful abstraction for storing unstructured data, as well as relational data with appropriate transformations (e.g., by mapping the primary key and a representation of the rest of the record's fields to the key and the value of the key-value pair, respectively [3, 35]). Internally, key-value pairs are stored in ordered maps to support consistent range operations, as shown later.

Figure 1 displays how our APIs enable SnapKV to support OLAP, OTLP, and machine learning (OLDA) workloads. OLAP, OLTP, and OLDA transactions must operate atomically, consistently, and isolated. Since these transactions are heterogeneous in their computation, we identify a set of common primitive operations and augment them with the capability to run other types of computations (e.g., machine learning inference, relational algebra). Among the common primitives, we also include our SNAPSHOT API, which allows programmers to specify multiple keys or ranges of keys (i.e., a snapshot). Importantly, the SNAPSHOT API is also managed by the concurrency control to guarantee its consistency with the other transactional APIs. This snapshot can be further manipulated using other transactional APIs, but more importantly it is used by the other types of computations to atomically retrieve and analyze the target snapshot.

At a high level, SnapKV implements a *snapshot isolation (SI)* [11] concurrency control through multi-versioning and validation of conflicts at commit time (more details in Section 3). We choose to support SI because, in addition to being widely used in commercial databases [33, 39] (although often called something other than snapshot isolation [11, 21]) and widely studied [12, 36], it provides real-time order, which

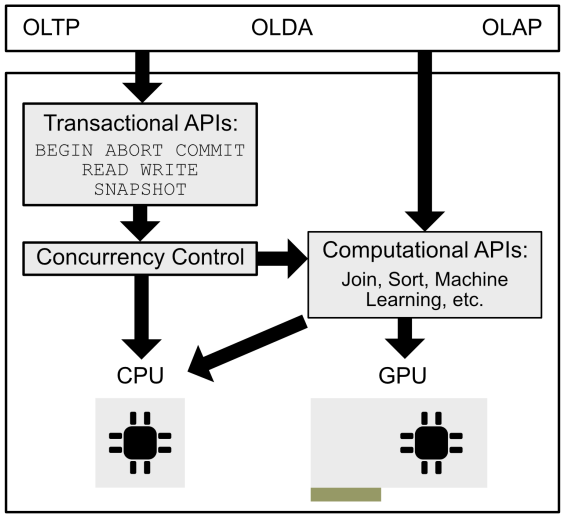


Figure 1: SnapKV's Heterogeneous Architecture for Heterogeneous Workloads.

is necessary for our workloads. Informally, real-time order guarantees that a transaction atomically occurs between invocation and commit and that all operations that committed before in time are visible. Within snapshot isolation, a global timestamp is used as the ordering mechanism.

SnapKV operates by receiving a `BEGIN` operation, at which point the transactional manager begins a transaction. The client can issue `READ` and `WRITE` or `SNAPSHOT` operations. The APIs work as follows:

- `READ`: takes a key or a range and returns a value or list of values.
- `WRITE`: takes a key-value pair to be written or deleted.
- `SNAPSHOT`: takes a list of keys and a list of ranges and returns the corresponding list of values.
- `ABORT`: used to intentionally abort a transaction.
- `COMMIT`: used to finalize a transaction.

The `SNAPSHOT` API implements a linearizable range query over multiple ordered maps and can be invoked within a transactional context. As opposed to the multi-get API available in other data repositories (e.g., Memcached [4]), our `SNAPSHOT` performs a single optimized traversal of the data repository, which is faster and critical to support long-running code (e.g., OLAP queries or transactions that use machine learning primitives).

To support heterogeneous computation within a transaction using the GPU, we extend the transactional programming model to allow running operations to program the GPU for efficient execution (e.g., CPU/GPU memory allocation and transfer, GPU kernel scheduling). In Figure 1, this is denoted by the computational API. Within a transaction, the computational APIs can be invoked on data provided by a transactional API (`READ` or `SNAPSHOT`) or can be invoked on data not stored within SnapKV (e.g. arguments to the transaction). Correctness is preserved by disallowing the GPU to modify data provided by `SNAPSHOT`, as well as synchronizing CPU and GPU execution to avoid parallelism within a transaction (more details in Section 4).

SnapKV does not limit its support of heterogeneous computation to what is provided within its pre-existing compute subsystem libraries; programmers can provide their own custom implementations of arbitrary operations (e.g., a linear regression task) to be run on both CPU and GPU. This heterogeneous computation can use SnapKV's primitives to retrieve data and manipulate data, consistently, with the only limitation that GPU tasks should not modify data. Customizability enables SnapKV to optimally adapt to new transactional workloads and run these operations on both the CPU and GPU.

2.1 Differing Approaches

The approach SnapKV takes is different from prior systems [13, 24, 35].

Figure 2 illustrates how existing HTAP [24, 35] and OLDA [13] systems utilize middleware or distributed protocols to couple together multiple nodes or services. This approach increases system complexity and often introduces undesirable trade-offs, such as the inability to guarantee real-time order across the system. On the other hand, SnapKV is built from the ground up and, in order to simplify system guarantees, it ensures SI across the key-value store. Furthermore, SnapKV is designed to support the computing capability of emerging hardware, such as GPUs, within transactions.

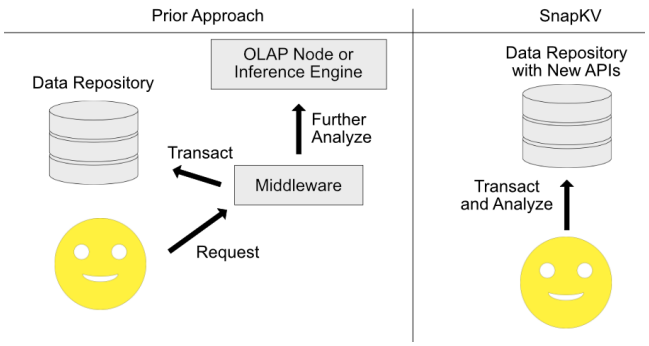


Figure 2: SnapKV vs Prior HTAP and OLDA Approaches.

The most basic differences in the systems can be seen when implementing a simple fraud detection microbenchmark on FeDB [13] versus SnapKV. Within our fraud detection, we want to store information about the current credit card transaction (e.g., user, location, etc.) and retrieve/pre-process some amount of prior transactional history. At this point, we can infer whether the current transaction is fraudulent or not. A

FeDB based implementation must run an insert transaction to insert a new record into a table, followed by another transaction to query the prior relevant transactions. These two operations are not atomic in FeDB. At this point, the current financial transaction and historical financial transactional information must be run through another server with an inference engine to infer.

Unlike FeDB, SnapKV does not require any networking or another node for inference. Also, this process would be representable as a single atomic operation; the transaction would begin with a `WRITE` followed by a `SNAPSHOT` of prior information. These operations would interact with the transactional subsystem and the concurrency control. Once the `SNAPSHOT` returns the relevant transactional history, the programmer will use the compute subsystem to copy the data to the GPU, run our machine learning kernels, and copy the data back. Once the compute subsystem finishes and copies the data, the programmer can handle the result of the inference and `COMMIT` through the transactional subsystem.

In Section 3 we detail how the transactional subsystem and the `SNAPSHOT` primitive works. In Section 4 we explain how the compute subsystem and heterogeneous computing is enabled within the key-value store. In Section 5 we detail our OLDA credit card fraud microbenchmark. Finally, in Section 6 we evaluate our approach in comparison to prior approaches.

3 Transactional Subsystem

SnapKV's transactional subsystem is designed to support the operations that are necessary to implement concurrent OLAP, OLTP, and OLDA workloads. This includes `READ`, `SNAPSHOT`, and `WRITE`. The two APIs retrieving data ensure the highest level of freshness (i.e., real-time order).

SnapKV's transactional subsystem supports SI through multi-versioning [36]. In SI, each read is executed as if taking a logical snapshot of the data repository at a given global time. Committing modifications to the data repository will increment a global timestamp and can only occur if no other concurrent transaction writes the same set of keys. Clients are able to interact with the SnapKV transactional manager through the six operations mentioned in Section 2. We determined that these operations have an affinity for the CPU instead of the GPU, and therefore we implement and optimize them for the CPU.

SnapKV implements an optimistic concurrency control (pseudocode shown in Algorithm 1) that ensures the Snapshot Isolation (SI) correctness level. It works as follows. A global timestamp is read on `BEGIN`. This is stored in `tx.timeStamp`. A map associates a key with a list of versioned values. When a `WRITE` occurs, it is buffered by the transaction manager (within the `tx.orderedWriteSet` in the pseudocode). When a `READ` occurs, `tx.orderedWriteSet` is checked to see if the key has been updated; otherwise the map is read, and the most recent version with a timestamp less than or equal to the transaction timestamp is returned. Recall that, because of the SI guarantees, the return values of read operations can be avoided to be recorded for future validation. `ABORT` frees the state of the transaction, while `COMMIT` initiates the commit protocol.

Algorithm 1 `COMMIT`

Require: Transaction *tx* and global timestamp *gts*

```
1:  versionLists ← { }
2:  for (k, v) ∈ tx.orderedWriteSet do
3:      l ← map.getVersionList(k)
4:      if l is null then
5:          l ← map.insertVersionList(k)
6:      end if
7:      lock_exclusive(l)
8:      versionLists.append(l)
9:      if l.latestTimestamp > tx.start then
10:         unlock(versionLists)
11:         return ABORT_AND_RETRY
12:     end if
13: end for
14: tx.commit ← ++gts
15: for l ∈ versionLists do
16:     l.prepend((tx.commit, tx.orderedWriteSet(l.key)))
17:     unlock(l)
18: end for
19: return COMMITTED
```

The commit protocol locks in exclusive mode the keys that will be written to in increasing order to prevent deadlock. If the key does not exist in SnapKV, a new version list will be inserted and marked with timestamp 0 (i.e., no read operation can retrieve it). While locking, the concurrency control validates that no concurrent transaction has written a newer version by comparing the transaction timestamp to the latest write of each key. If this does not hold, the transaction aborts and retries; otherwise the global timestamp is atomically incremented, updated values are written, and locks are released.

Upon commit, SnapKV must modify the in-memory storage. To implement the storage in practice, we rely on a concurrent map that implements an elementwise linearizable skiplist [1]. It should be noted that SnapKV is not limited to using this map; it is possible to use any linearizable mapping data structure that supports reads, modifications, and concurrent iteration over the data structure.

The concurrency control also supports calling GPU kernels through the compute subsystem (more details on how this interacts with the concurrency control are in Section 4.1).

Our main innovation to the transactional manager is the addition of the `SNAPSHOT` primitive (detailed in Section 3.1). This primitive is able to provide a method to perform a large series of real-time order reads, enable a programmer to

quickly move the resulting data to a GPU, and help minimize the data structure's lock holding time.

3.1 SNAPSHOT Primitive

Within a data repository, there are two different types of reads that are important to support: range queries and point queries. The SNAPSHOT primitive implements these reads as a unified primitive.

To call SNAPSHOT, a list of points and ranges is taken as an argument. The transactional subsystem will iterate through this list of arguments and perform each range query or point query. The transactional subsystem will then return the keys and values associated with each read. For example: the call SNAPSHOT({a,range-b}) will return a list containing the value of a followed by the first value in range-b, the second value in range-b, and so on, along with the associated keys. There is also an index that is returned that contains the start location and end location of the *i*th argument in SNAPSHOT.

To implement this primitive in practice, SnapKV uses the multi-version concurrency control and a concurrent map which is elementwise linearizable. Algorithm 2 details how the SNAPSHOT primitive works. For each argument given to SNAPSHOT, the individual query is run. For any point query passed to snapshot, SnapKV performs a READ and appends it to an array of keys and values. For range queries, the maps are traversed in an elementwise linearizable fashion. At each element, SnapKV acquires a shared lock on the version list and get the value associated with the timestamp of the transaction (i.e., the value with a timestamp less than or equal to the start timestamp of the transaction). When performing a range query in the absence of concurrent updates, it is guaranteed it will always find the latest version consistent with the transaction's start timestamp.

If there is a COMMIT concurrent with a range query, and it increments the global timestamp before the transaction issuing the range query reads the timestamp, then all updates in the COMMIT must be observed by the range query. SnapKV guarantees this through the multi-versioning scheme along with the lock protection on the version lists. Specifically, since the COMMIT acquires all the locks before incrementing the global timestamp, the range query will be blocked if needed until the new versions are added.

In the case that the transaction issuing the range query reads the timestamp before the COMMIT increments the global timestamp, even if it will be blocked by the COMMIT it will not observe those updates because it will have a larger timestamp.

3.2 READ vs SNAPSHOT

SnapKV's SNAPSHOT primitive is advantageous over READ by enabling ease of copying data when doing heterogeneous programming. The structure of the vectors returned by the SNAPSHOT primitive is designed to facilitate offloading future computations to the GPU. Since it returns vectors of keys, values, and indexes, it is straightforward to move these to the GPU with three memory copies. If SNAPSHOT was not used, individual copies to the GPU must be done or the programmer must assemble the vector in their transaction and perform a similar copy. By using SNAPSHOT we are able to better support the use of heterogeneous architectures in SnapKV's compute subsystem (more details in Section 4.1).

Algorithm 2 SNAPSHOT primitive

```
Require: a set Args of points and ranges
keys ← vector({})
values ← vector({})
index ← vector({})
count = 0
for arg ∈ Args do
    if arg is range query then
        l = map.iterator(arg.begin)      ▷ l is a version list
        for l != map.iterator(arg.end); ++l do
            lock_shared(l)
            kv ← l.getKVAtTime(currentTx.start)
            unlock_shared(l)
            keys.append(kv.key)
            values.append(kv.value)
            count++
        end for
        iter = writeSet.iterator(arg.begin)
        for iter != writeSet.iterator(arg.end); ++iter do
            if iter.key ∈ keys then
                values.update(value)
            else
                keys.append(iter.key)
                values.append(iter.value)
                count++
            end if
        end for
        index.append(count)
    else
        kv ← writeSet(arg.key)
        if arg.key ∉ writeSet then
            lock_shared(l)
            kv ← l.getKVAtTime(arg, currentTx.start))
            unlock_shared(l)
        end if
        keys.append(kv.key)
        values.append(kv.value)
        index.append(++count)
    end if
end for
return {keys, values, index}
```

SNAPSHOT can also be advantageous for performing index joins when query selectivity, i.e., the probability a key will be selected when selecting keys based on a predicate, is high.

This case occurs in `Q3` of ch-benCHmark [14], an HTAP benchmark, where there are index joins. A database must join `customer`, `order`, `neworder`, and `orderline` where the customer's state matches a predicate and then perform aggregation. Traditional query optimizers like TiDB's optimizer [24] approach this by joining `order` with `customer` and filter out customers on this join. At this point, the cardinality may be less than the cardinality of the `order` table, and the database may be able to perform the rest of the operations with fewer reads.

SnapKV can implement the above join in two ways. In both ways, it represents tables as key-value pairs with tuple `(table name, primary key)` as the key and the record as the value. When attempting to benefit from low selectivity, SnapKV would use `READ` and perform an index join, which is comprised of an $O(\log n)$ operation on the skiplist for each element in the range.

When selectivity is high, this means that the cardinality of `order` joined with `customer` after the filtering out customers is close to the cardinality of `order` joined with `customer`. This limits the benefit of approaching the query this way. Using `SNAPSHOT` instead, SnapKV will get all tables and use the compute subsystem to perform the joins. When there is minimal impact of cardinality, this means performing an $O(n)$ traversal over the skiplist. Comparatively, not using `SNAPSHOT` would require re-traversing the skiplist, which is $O(\log n)$ for each entry. This repeated traversal would lead to $O(n \log n)$ operations for the index join.

In Section 6 we evaluate this approach to using `SNAPSHOT` and find it can lead to up to about 20% increase in throughput to approach joins this way when there is high selectivity. In order to efficiently use the primitive, the programmer must make a determination whether to perform a snapshot or individual reads. We approach the decision to use `SNAPSHOT` and `READ` based on our insights gained from testing performance versus selectivity in Section 6, which suggests above 80% selectivity is best for using `SNAPSHOT`. We also advise grouping any reads that follow one another as a snapshot when programming.

When implementing ch-benCHmark, we generally utilize `SNAPSHOT` on the queries and `READ` on the transactions. In the case of our credit card fraud benchmark (described in Section 5), we found ourselves using `SNAPSHOT` since the credit card transaction logging and detection operation has initial reads.

4 Compute Subsystem

The compute subsystem enables programmers to accelerate complex data analysis and operations within SnapKV by using both the CPU and GPU. In order to achieve the high performance and scalability goals of the compute subsystem, SnapKV enables GPU compute within transactions and builds upon GPU asynchronous programming APIs. Details about this are in Section 4.1.

Another important characteristic of SnapKV is that transactions are compiled as procedures. Within these procedures, a programmer can utilize CPU primitives (e.g., join) and tell SnapKV to dynamically load user specified kernels or functions. We detail this in Section 4.2.

4.1 GPU Compute within Transactions

Supporting GPU compute within transactions presents two problems, how the CPU and the GPU are able to share memory and how to maintain high performance when switching execution between the CPU and the GPU. Consider implementing a GPU accelerated fraud detection transactional benchmark within SnapKV. In this benchmark, historical financial transactional data are read, GPU accelerated machine learning is then used to infer whether the new transaction is fraudulent, and finalize the new financial transaction.

In order to enable the GPU to work on transactional data, the data must be accessible by the GPU and the results visible to the CPU. SnapKV relies on the standard memory copy mechanism [2] between host memory and device memory, but to retain high performance and limit overheads, it ensures minimal synchronization between the CPU and GPU. That is done by utilizing non-blocking queues of operations (called streams [23] in CUDA) to asynchronously schedule kernels for the runtime to execute.

Memory access between the CPU and GPU should be done efficiently to enable high-performance processing of transactional data. While the CPU can directly access data through the transactional manager because it is locally stored, the GPU cannot since the data structures are located in CPU memory and written by the CPU only. To overcome this limitation, the CPU should move the relevant key-value pairs to the GPU. Likewise, any data the GPU should return to the CPU, will be written into its device memory and copied back to the CPU.

SnapKV utilizes CUDA streams, which are queues of kernel invocations and memory movement to be executed by the runtime. SnapKV's allows transactions to include CPU and GPU code blocks, which enables the invocation of CUDA stream to enqueue GPU operations. Within a CPU code block, only CPU code can be called (e.g., transactional APIs, CPU join implementations); within a GPU code block, only GPU kernels and APIs may be called within the context of the stream (e.g., deep learning kernels or memory movement). At the end of a GPU code block, the CPU is synchronized with the GPU. In our implementation, we rely on the programmer to write their transactions in this blocked manner. As an example, we implement our fraud detection benchmark as a sequence of the three code blocks. The first is a CPU block that starts a transaction and calls `SNAPSHOT` within it. This is followed by a GPU code block where the results of this `SNAPSHOT` are copied to the GPU, machine learning is

run, and the results are copied back. Following the GPU code block is a CPU code block to finalize the transaction and `COMMIT`. All the code within the GPU code block occurs within the context of a single stream.

The above approach forbids parallelism between the CPU and GPU, which is important to maintain transactional semantics. In our fraud detection benchmark, for example, without synchronizing after the movement of the results from the GPU to the CPU, the CPU could miss important outcomes produced by the GPU computation. Importantly, our approach avoids synchronizing the CPU and GPU after every GPU call, which would lead to poor performance [32]. It also enables concurrent transactions to concurrently utilize the GPU and CPU through its use of streams, which leads to high performance.

SnapKV further relies on GPU kernels to be equivalent to a single thread execution on the CPU. With this guarantee, we can reliably have the same semantics as a CPU-only data repository.

4.2 Compiling Transactions and Analytics

In SnapKV, transactions are written in C++, compiled as procedures that can be dynamically loaded as kernels and functions. In order to add new kernels, SnapKV compiles with the CUDA compiler `nvcc` and the C++ compiler and creates a shared object file. The GPU kernels will be in a fat binary in the shared object file. When loading the transactions, SnapKV utilizes the operating system's existing dynamic library loading functionality. After the functions are loaded, SnapKV searches through the shared object file for the fat binary. Once SnapKV finds the fat binary the programmer wants to load, SnapKV uses the CUDA driver API to load in the kernels. The CPU primitives are added within C++ header files as inlined, and often templated functions. By inlining, SnapKV is also able to potentially help gain performance and benefit from further compiler optimization.

By compiling transactions as procedures, SnapKV enables handcrafted solutions to address specific workload requirements, as shown in our evaluation (Section 6).

5 Online Decision Augmentation Fraud Microbenchmark

Our Online Decision Augmentation (OLDA) microbenchmark is based off of the credit card fraud problem presented in FeDB [13]. Since the benchmark used in FeDB [13] is not open, we seek to create a synthetic benchmark to test OLDA, which consists of feature-engineering, machine learning inference, and an insertion into a data repository.

The goal of this microbenchmark is to stress multiple pieces of an OLDA system. This includes utilizing heterogeneous architectures to accelerate inference processing, which have been explored in practice on anonymized or private data sets [7, 9, 40]. We specifically evaluate by performing inference on a GPU to expose the costs of moving data between devices, as well as solely relying on slower single-precision floating-point operations. To that end, we also aim to ensure the microbenchmark performs the same operations regardless of whether a transaction is detected as fraudulent, to guarantee that accuracy is not an issue. There are opportunities to utilize lower precision computation, including half-precision, 8-bit floating point, or microscaling to further accelerate performance, but we make no assumptions on hardware support.

We design our benchmark depicting the scenario in which users perform financial transactions and a database is used to store relevant information for inferring credit card fraud. Along with each transaction, the following metadata is also provided: the latitude and longitude of the location where the transaction is performed, an identifier of the user, and an identifier of the database transaction. The microbenchmark distinguishes fraudulent and non-fradulent credit card transactions depending on their geographic location. Transactions are considered non-fraudulent if they are issued within 300km of a user's prior transaction(s).

The benchmark is parameterized by the number of users issuing financial transactions. To set up the microbenchmark, historical credit card transactions for users are generated with a uniform random distribution and initially located uniformly around four locations. When benchmarking, each credit card transaction is normally distributed around the associated location, with a standard deviation of 1 in both longitude and latitude and no co-variance between the longitude and latitude.

Each time a credit card transaction is received by a database, the following steps must occur:

- The prior credit card transaction(s) for a user is retrieved;
- The distance between the prior and current credit card transactions is calculated;
- The features: distance, latitudes, and longitudes are fed to a dense neural network trained on the data for inference;
- A new credit card transaction for a user with the associated latitude and longitude is inserted into the database.

These steps are done to test a typical OLDA workload, in which concurrent writes, reads, feature engineering, and inference are performed.

We initially train our neural network on the synthetic data set. The neural network consists of single precision floating point layers with a 5 element input, followed by a 128 neuron layer and a ReLU. This is then output into a 2 neuron layer with a sigmoid.

SnapKV performs these operations as a single transaction, with the support of GPU acceleration. Other competitors, such as FeDB [13], execute these operations as a read transaction, a

separate inference process, and then an insert transaction. The absence of atomicity between these operations could impact the identification of fraudulent transactions.

6 Evaluation

To assess SnapKV's performance under OLDA workload, we contrast SnapKV with FeDB [5, 13] (a state-of-art OLDA database) using the credit card fraud detection microbenchmark described in Section 5. On the other hand, to evaluate SnapKV's performance under HTAP workloads, we compare it with two HTAP databases, Vegito [3, 35] and TiDB [24], using the ch-benCHmark [14] benchmark. ch-benCHmark combines the well-known TPC-C [15] benchmark with the TPC-H [16] benchmark to model an application of HTAP for a warehouse business.

SnapKV is built with GCC-10 and CUDA 11.6. We also utilize CUTLASS [26] version 2.9.0 for our deep learning kernels.

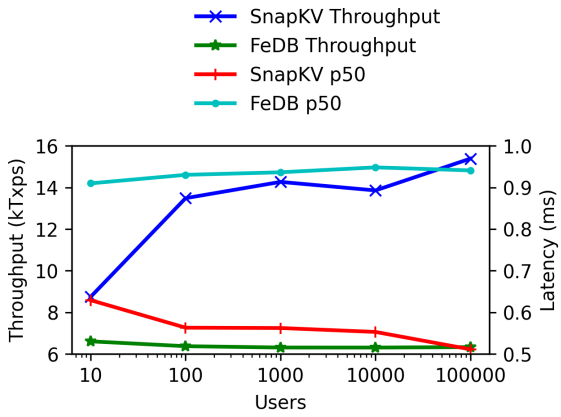
6.1 OLDA Workload

The test bed for our OLDA credit card fraud microbenchmark is a Cloudlab R7525 [18], which has two 32-core AMD Epyc 7542 and two NVIDIA Tesla V100. We only utilize one V100 and pin threads to one socket for the experiments. On the CPU, we fix the number of threads to 8 because in that configuration FeDB performs best in comparison to SnapKV.

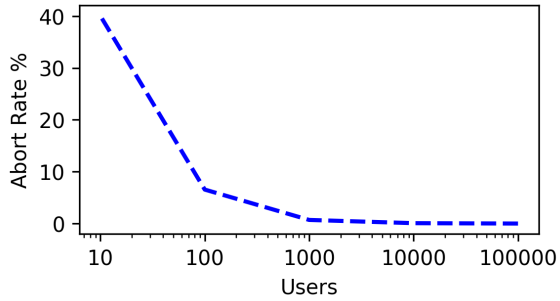
When we analyze the performance of SnapKV we find that it is able to achieve 80% utilization of the GPU starting at 2 threads as reported by nvidia-smi. We can attribute this to the use of multiple streams, while the GPU is executing inference for some transaction, it can continue to process other transactions on the CPU. Since the benchmark depends on executing on the GPU before writing back to the database, SnapKV is effectively bound by the inference computation on the GPU, and increasing CPU threads will not improve performance.

We begin evaluating SnapKV on OLDA workloads by varying the number of credit card users (Figure 3). In Figure 3a we show both the throughput and the latency. As can be seen, we consistently achieve higher throughput and lower latency than FeDB. We are able to gain increased throughput through combining both our inference and transactional systems together. FeDB utilizes HTTP requests to run queries to perform its feature engineering. This is then followed by the GPU accelerated deep neural network inference, and then a second transaction to write the credit card transaction. In contrast, we limit memory movement from system to system, and by doing so are able to achieve high performance while providing transactional semantics. We achieve between 1.32 and 2.43x speedup and 31-48% improvement in latency.

Looking more closely at the contention level in the workload, at 10 users we find that there is a significant number of



(a) Throughput and Latency when Varying Credit Card Users (Throughput on the Left Axis and Latency on the Right Axis).



(b) SnapKV Abort Rate when Varying Credit Card Users.

Figure 3: Performance when Varying Credit Card Users in Credit Card Microbenchmark.

conflicts, which leads to a high abort rate (40% of transactions are retried) and high system latency (0.63ms). Also, increasing the number of users decreases aborts. This trend is visible in Figure 3b. At 1,000 users, throughput and latency begin to stabilize as the number of users in the system increases. We find that at 10,000 users the contention level is acceptable, and so we will use this configuration in future experiments. While the abort rate is high for SnapKV, by minimizing the cost of moving data between the CPU and GPU and integrating inference support within the data repository it is able to outperform FeDB.

SnapKV is not only able to utilize the GPU to improve performance of OLDA workloads, but it is also able to utilize the `SNAPSHOT` primitive.

We now evaluate the performance of the `SNAPSHOT` primitive. Specifically, we consider how snapshot size impacts performance, and we do this by varying the number of historical transactions retrieved. Results are shown in Figure 4.

When moving from no additional snapshot size to 100 extra historical transactions, we observe a 7.4% increase in p50 latency and parity with transactional throughput. This suggests that there is minimal overhead to `SNAPSHOT` in the range of

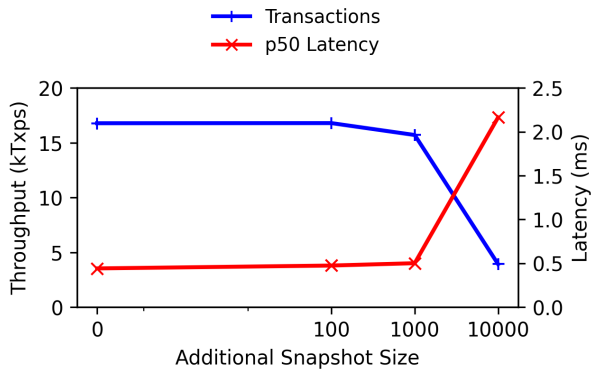


Figure 4: Varying Size of Additional Key-Value Pairs Snapshotted in OLDA Microbenchmark.

100 records. Increasing up to 1,000 reads per transaction, we find continued minimal difference in performance, maintaining 93.5% of transactional performance with a 5.6% increase in latency. It is only when we increase to 10,000 reads in a single transaction that we experience a drop in performance.

6.2 OLAP and OLTP Workload

Our OLTP and OLAP competitors, TiDB and Vegito, utilize a testbed with 2 Intel Xeon Platinum 8160 CPU with hyper-threading enabled, giving us 48 cores with 96 hardware threads, and a Mellanox Connect X-5 for loopback. SnapKV utilizes the `SNAPSHOT` operation and a optimistic concurrency control to minimize the duration during which contention occurs between transactions. On the other hand, in order to achieve desirable performance for concurrent OLAP and OLTP transactions, TiDB and Vegito utilize the existing replication mechanism to execute OLAP queries in separate processes.

We test the competitors with multiple processes on a single node and rely on the loopback network interface to ensure the performance differences between all competitors are not due to the networking. We also only utilize the CPU for executing queries on SnapKV to minimize differences due to GPU vs CPU acceleration. Vegito relies on Intel TSX-NI and a Mellanox RDMA network card, so it is not possible to reproduce results on the AMD Epyc system. We reproduce TiDB's results on the AMD Epyc system and find it consistently performs better on the Intel Xeon machine, so we only compare to the Intel Xeon machine. This configuration enables Vegito to offload memory copies between processes to the RDMA network card, effectively treating the card as a hardware accelerator.

When running our experiments with ch-benCHmark, we run with 10 warehouses. The version of TiDB used is 5.4.0. TiDB is run with 3 TiKV processes, 3 placement drivers, and 2 TiDB processes. Unless otherwise mentioned, Vegito is configured to broadcast the establishment of a new epoch every 15ms, meaning that it could take longer than 15ms to establish the next interval when writes are visible on the OLAP process. We run Vegito with 1 primary process, 1 backup process, and 1 OLAP process.

To implement ch-benCHmark in SnapKV, we utilize key-value tables with primary keys mapping to rows. In this workload, we run 90% `New Order` transactions followed by 10% `Delivery` transactions. Both `New Order` and `Delivery` transaction profiles write to the `orderline` and `neworder` tables. We also concurrently run the `Q1` query profile, which reads from the `orderline` table to perform aggregation over delivered orders (orders modified by the `Delivery` transaction). This profile creates dependencies and anti-dependencies between all transactions and queries, hence making the level of data freshness critical. We generate a uniform random warehouse and district (1 of 10 districts) for each transaction, and will read from `orderline` and `neworder` in `Q3` when a customer state matches a query. Unless otherwise stated the probability a customer is selected is 1.7%. In order to perform queries, we utilize single threaded transactions. Instead, TiDB and Vegito parallelize parts of their queries. Unlike our competitors, we are able to run transactions and analytics with real-time order guarantees using ch-benCHmark.

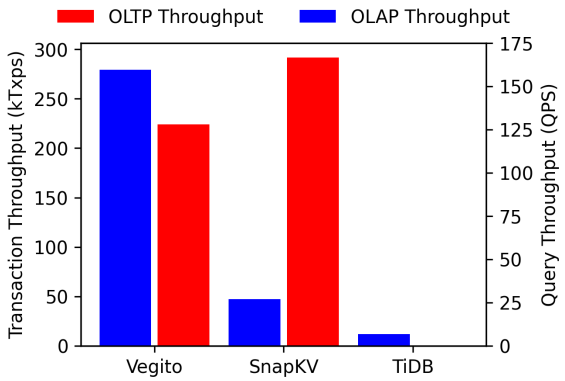


Figure 5: Throughput of different HTAP Solutions.

In Figure 5, we run our custom `New Order`, `Delivery`, and `Q1` workload. As also confirmed in [35], TiDB does not perform as well as in-memory systems due to the disk access costs (to minimize that, our testbed is equipped with SSD drive). SnapKV is able to achieve a significant improvement in transactional performance (up to 600x) and a 4x improvement in query performance.

Compared to Vegito, SnapKV obtains 1.3x the transactional throughput, but it has a lower query throughput, up to 16.8% of Vegito's. The improvement in transactional throughput comes from the minimized concurrency control overhead: SnapKV does not need to log to a backup node upon commit, while Vegito does. On the other hand, Vegito is able to perform better in terms of query throughput because of

two reasons. First, it works on stale data, therefore no synchronization with the ongoing workload is required; second, many of its components are optimized for a column store relational database. As opposed to Vegito, SnapKV guarantees real-time order and does not focus on implementing a column store. Future work could be spent considering this optimization.

In our next experiment (results in Figure 6) we dig deeper in the performance impact of allowing stale accesses. We exclude TiDB because of its comparatively poor performance.

Vegito maintains an oracle that periodically broadcasts the establishment of a new epoch in which the analytical database apply batch updates. In this experiment, we vary the epoch time. We calculate freshness as the number of epochs difference between the OLAP node and the OLTP node at any given time. To achieve real-time order, the epochs would need to be consistently equal.

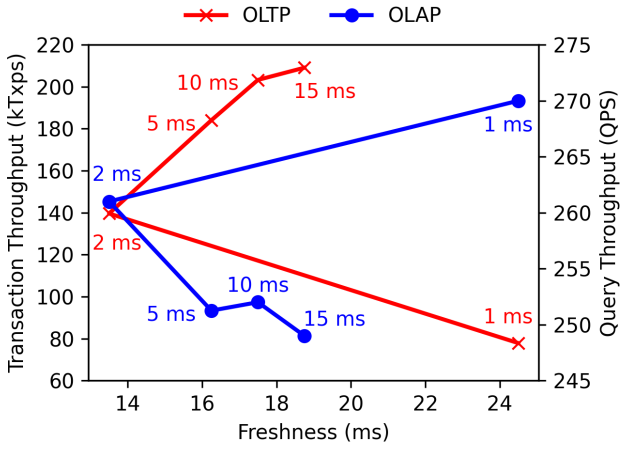


Figure 6: Throughput versus Freshness of Vegito While Varying Epoch Timing (Epoch Times Written on Plot). SnapKV has 0ms freshness, 292 kTxps, and 26.9 QPS.

By varying the epoch time, we find Vegito is unable to reach real time order (0ms). Vegito is only able to achieve a minimum freshness of 13.5 ms when the epoch time is set to 2ms. When running at its most fresh, Vegito is only able to perform 66.8% of the maximum transactional throughput and 96.7% of the maximum query throughput it can achieve.

Moving from a 15ms epoch to a 2ms epoch, Vegito is able to improve freshness, however, query throughput increases and transactional throughput decreases. That is because, decreasing the delays between epochs entails the transactional subsystem needs to push all logs to the backups at higher rate. As a result, more synchronization is needed to log updates and transactional performance suffers. Interestingly, in this case the OLAP process is able to serve more requests while waiting for updated logs to be pushed. This increases the query throughput and keeps it stable; query throughput only varies between 249 and 270 qps while transactional throughput varies between 78 ktxps and 209 ktxps.

Vegito is unable to maintain its freshness when moving from a 2ms epoch to 1ms epoch. At this point, logging to the OLAP process is a significant overhead for the OLTP process. This overhead leads to a significant skew between the epoch that the OLAP process has stored and the epoch that the OLTP process has stored, which makes the OLAP transactions more stale. Unlike Vegito, SnapKV is able to maintain consistency for both the transactions and queries without significant impact on their performance.

Next, we evaluate the performance of the `SNAPSHOT` primitives versus the `READ` primitive when changing the selectivity of queries (i.e., the probability a key is selected). When using `READ`, we perform index joins and utilize the selectivity of the query to minimize the number of reads that must occur of the following tables. When using `SNAPSHOT`, we snapshot all the tables, rebuild indexes, and perform the joins. In Figure 7, we evaluate changing the selectivity of ch-benCHmark's `Q3` query by changing the proportion of customers we filter out by selecting on the customer state when implementing with `READ` and when implementing with `SNAPSHOT`.

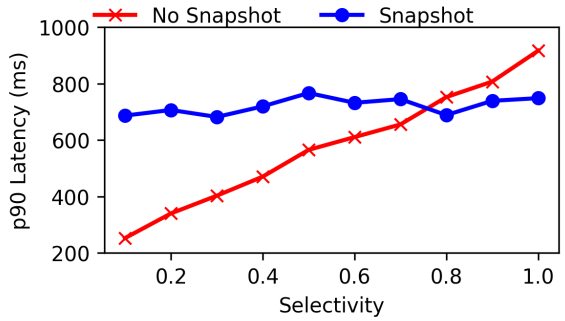


Figure 7: Read Versus Snapshot (p90 Latency) While Varying Selectivity of the `Q3` Query.

At a selectivity of 0.1, the query throughput when using `SNAPSHOT` is 34% of using `READ`s and the p90 latency is 2.7x that of using just `READ`s. At 0.8 selectivity and above, we find that `SNAPSHOT` becomes a better option to achieve high performance transactions. It is able to achieve 6% greater throughput and 8.5% of the p90 latency. This is because we are able to perform all reads in a sequence, which limits time for contention. We also are able to benefit from having a copy of data, which we can reuse rather than needing to perform reads on the same key multiple times. In the best case, using `SNAPSHOT` can result in 20.4% greater throughput and a 18.3% decrease in the p90 latency.

7 Related Work

Both academia and industry have been interested in the prospect of combining various analytical workloads and trans-

actions in a single data repository [8, 10, 13, 24, 27–29, 31, 35, 37]. MemSQL [37], SQLServer [29], TiDB [24], Vegito [35], and FeDB [13] support these kinds of workloads. These analytics and transactions are not limited to relational or key-value models, and include graph processing [34, 41] and in-database machine learning [5, 13].

Recent innovations in HTAP databases have focused on using replicas to serve queries over fresh data [8,24,35]. TiDB and Vegito are recent examples of utilizing the availability mechanism to replicate state onto a query processing node. Unlike these solutions, we only rely on a single node/process to provide support for queries and transactions.

Traditional solutions for HTAP focus on multiversion concurrency control. Diva [27] focuses on supporting HTAP in disk based MVCC databases. HyPer [31], RateupDB [28], Vegito [35] and others utilize aspects multi-versioning as well. We similarly focus on utilizing multiversioning, which enables the optimistic approach to concurrency control that we rely on in SnapKV.

Other HTAP database solutions, such as RateupDB [28] have focused on using an alpha and delta store model, where transactions modify the delta storage and are merged over time into the alpha store. RateupDB also focuses on the prospect of using the GPU for analytical processing similar to SnapKV. Unlike SnapKV, RateupDB is specifically designed for the relational model, relational algebra, and column storage. SnapKV takes a more general approach, allowing for any computation on the GPU within transactions.

Key-value stores enable a simpler data model and have been greatly explored. Recent publications have focused on enabling key-value stores to benefit from new memory architectures, such as how ChameleonDB [46] is capable of utilizing persistent memory. EvenDB [19] focuses on exploiting spatial locality within key-value workloads to enable high-performance. SnapKV neither benefits from new memory architectures nor exploits spatial locality, but these techniques are complementary to SnapKV.

Even more related papers on key-value stores focused on the prospect of GPU computing. KVCG [17] provides a method to serve skewed key-value store workloads by cooperatively utilizing the CPU and GPU. MegaKV [45] provides another method for serving key-value store workloads using the GPU as an accelerator. Unlike KVCG and MegaKV, we focus on transactional workloads and must ensure atomicity, isolation, and consistency across more than one operation. We also do not utilize the GPU to do storage, only computation.

Related key-value store works enable multi-get operations, where a single request can read multiple keys in an elementwise-linearizable fashion. Key value stores such as Memcached [4] and KVCG [17] enable multi-get operations. `SNAPSHOT` is similar to multi-get in that it can get multiple keys by providing a set of keys or a range. Unlike multi-get, `SNAPSHOT` operations are used within transactions and guarantee consistency between all of the reads in the snapshot. `SNAPSHOT` is designed as an API for optimizing read-only and read-heavy transactional workloads.

For the programability of our system, we rely on existing libraries to implement queries. We utilize C++ and the CUDA programming language [2] and provide support for libraries like CUTLASS [26] to design of deep learning solutions. We demonstrate the ability to run deep learning inference in our key-value store. We envision other CUDA libraries may be of use in our system, and could be easily used within SnapKV. Related to the deep-learning aspects of our work, Han et al. [22] present a GPU preemption methodology to improve throughput for inference workloads with real-time deadlines. This work is complementary to our work. Our methodology does not focus on preemption and instead focuses on evaluating the feasibility and performance of using the current CUDA runtime to compute within a key-value store.

Our closest related works support both transactions and analytics. TiDB [24] and Vegito [35] use the availability mechanism to run analytics on a separate node. TiDB performs best on fast storage like NVMe. Vegito, however, is in memory and utilizes DrTM+H [42] to support transactional workloads with remote direct memory access (RDMA). The backup analytical node is based on MonetDB [25]. Both TiDB and Vegito have a staleness issue (order of tens of *ms* in Vegito), which makes it more difficult to reason about the system, and prevents these systems from being broadly applied.

FeDB [13] is a closely related custom solution for feature engineering and machine learning inference. It is designed in contrast to HTAP solutions like TiDB and Vegito in how it needs even fresher data. Similar to FeDB we support machine learning and provide the programmer with the freshest possible state. Unlike FeDB, SnapKV is designed as a general solution, not just a solution for online decision augmentation workloads.

8 Conclusion

We introduced SnapKV, a transactional data repository architecture for heterogeneous workloads (OLTP, OLAP, and OLDA) that guarantees real-time order without sacrificing performance. This is achieved through the use of the new `SNAPSHOT` API and by supporting heterogeneous devices. Our evaluation confirms that SnapKV is able to outperform state of the art competitors in both OLDA and HTAP workloads.

References

[1] Intel® threading building blocks developer guide. URL: https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-guide.html.

[2] Cuda toolkit documentation, 2018. URL: https://docs.nvidia.com/cuda/archive/9.1/.

[3] Vegito: a fast distributed in-memory htap system, Dec 2021. URL: https://github.com/SJTU-IPADS/vegito.

[4] Memcached, October 2022. URL: https://github.com/memcached/memcached.

[5] 4Paradigm. Openmldb, May 2022. URL: https://github.com/4paradigm/OpenMLDB.

[6] 4Paradigm. Intelligent banking, January 2023. URL: https://en.4paradigm.com/industry/banking.html.

[7] Youness Abakarim, Mohamed Lahby, and Abdelbaki Attioui. An efficient real time model for credit card fraud detection based on deep learning. In *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*, SITA'18, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3289402.3289530.

[8] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. Proteus: Autonomous adaptive storage for mixed workloads. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 700–714. ACM, 2022. https://doi.org/10.1145/3514221.3517834.

[9] Fawaz Khaled Alarfaj, Iqra Malik, Hikmat Ullah Khan, Naif Almusallam, Muhammad Ramzan, and Muzamil Ahmed. Credit card fraud detection using state-of-the-art machine learning and deep learning algorithms. *IEEE Access*, 10:39700–39715, 2022. https://doi.org/10.1109/ACCESS.2022.3166891.

[10] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf.

[11] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[12] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *Journal of the ACM (JACM)*, 65(2):1–41, 2018.

[13] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory. *Proc. VLDB Endow.*, 14(5):799–812, 2021.

[14] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload ch-benchmark. In *DBTest '11*. ACM, 2011.

[15] Transaction Processing Performance Council. Tpc benchmark c revision 5.11, February 2010.

[16] Transaction Processing Performance Council. Tpc benchmark h revision 3.0.1, April 2022.

[17] dePaul Miller, Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. KVCG: a heterogeneous key-value store for skewed workloads. In Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz, editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 5:1–5:12. ACM, 2021. https://doi.org/10.1145/3456727.3463779.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/duplyakin.

[19] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: optimizing key-value storage for spatial locality. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 27:1–27:16. ACM, 2020. https://doi.org/10.1145/3342195.3387523.

[20] Google. Introducing alloydb for postgresql: Free yourself from expensive, legacy databases, May 2022. URL: https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql.

[21] The PostgreSQL Global Development Group. Transaction isolation, November 2022. URL: https://www.postgresql.org/docs/current/transaction-iso.html.

[22] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *OSDI 22*, pages 539–558.

[23] Mark Harris. Gpu pro tip: Cuda 7 streams simplify concurrency, January 2015. URL: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, 2020.

[25] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012. URL: http://sites.computer.org/debull/A12mar/monetdb.pdf.

[26] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, Duane Merrill, Aniket Shivam, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Matt Nicely. CUTLASS. URL: https://github.com/NVIDIA/cutlass.

[27] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. Diva: Making mvcc systems htap-friendly. In *SIGMOD '22*, page 49–64. ACM, 2022.

[28] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The art of balance: A rateupdb experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.*, 14(12):2999–3013, 2021.

[29] Microsoft. Microsoft data platform. URL: https://www.microsoft.com/en-us/sql-server.

[30] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 368–384, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3503221.3508412.

[31] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM, 2015. https://doi.org/10.1145/2723372.2749436.

[32] NVIDIA. Cuda c++ best practices guide, March 2022. URL: https://docs.nvidia.com/cuda/archive/11.6.2/cuda-c-best-practices-guide/index.html.

[33] Oracle. Oracle database 19c, January 2023. URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/index.html.

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

[35] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 219–238. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/shen.

[36] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. URL: https://www.db-book.com/.

[37] SingleStore. Singlestore. URL: https://www.singlestore.com.

[38] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A dbms-oriented operating system. *Proc. VLDB Endow.*, 15(1):21–30, 2021. URL: http://www.vldb.org/pvldb/vol15/p21-skiadopoulos.pdf, https://doi.org/10.14778/3485450.3485454.

[39] Michael Stonebraker and Lawrence A Rowe. The design of postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[40] Sumaya S. Sulaiman, Ibraheem Nadher, and Sarab M. Hameed. Credit card fraud detection using improved deep learning models. *Computers, Materials and Continua*, 78(1):1049–1069, 2024. URL: https://www.sciencedirect.com/science/article/pii/S1546221824001619, https://doi.org/https://doi.org/10.32604/cmc.2023.046051.

[41] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, and Lei Chen. Galaxybase: A high performance native distributed graph database for HTAP. *Proc. VLDB Endow.*, 17(12):3893–3905, 2024. URL: https://www.vldb.org/pvldb/vol17/p3893-tong.pdf.

[42] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/wei.

[43] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 31–46, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3437801.3441602.

[44] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeffrey F. Naughton, and John Cieslewicz. F1 lightning: HTAP as a service. *Proc. VLDB Endow.*, 13(12):3313–3325, 2020.

[45] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, 2015.

[46] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 194–209. ACM, 2021. https://doi.org/10.1145/3447786.3456237.