Karst: Transactional Data Ingestion Without Blocking on a Scalable Architecture

Zhifang Li, Beicheng Peng, Qiuli Huang, and Chuliang Weng

Abstract—Although real-time analytics on the up-to-date dataset has become an emerging demand, many big data systems are still designed for offline analytics. Particularly, for critical applications like Fintech, *transactional data ingestion* ensures a timely, always-correct, and scalable dataset. To carry out append-only ingestion, existing OLTP/HTAP systems are based on strict transactions with imperfect scalability, while NoSQL-like systems support scalable but relaxed transactions. How to ensure essential transactional guarantees without harming scalability seems to be a non-trivial issue. This paper proposes *Karst* to bring transactional data ingestion for existing offline analytics. We notice that blocking *two-phase commit* (2PC) to resolve transactional data ingestion is a performance killer for the partitioned analytical systems. Karst introduces a scalable protocol called *metadata-oriented commit* (MOC) that converts each distributed transaction into multiple partial transactions to avoid 2PC. Moreover, to ingest massive data into plenty of partitions, Karst also employs lazy persistence, lightweight logging, and optimized data traffic. In experiments, Karst could achieve up to about 2x∼10x performance over relevant systems and also shows remarkable scalability.

Index Terms—Data ingestion, real-time analytics, distributed transaction, two-phase commit

1 Introduction

Since the rise of time-sensitive and data-intensive applications, OLAP systems face challenges in timeliness, correctness, and scalability. Taking Fintech as an example, real-time trade recommendations [1], [2] improve investments by extracting past patterns and current market conditions from an up-to-date database, in which endless records of stock exchange are continuously inserted. To handle massive data, the database divides contained records into multiple *partitions* among the cluster to scale with the growing number of CPU cores [3]. Meanwhile, analysts could submit the OLAP query to analyze both historical and recently-ingested data. Under concurrent data ingestion, the transaction mechanism helps to ensure serializable isolation, where the query runs on an up-to-date and consistent dataset.

Currently, many big data systems, such as Hadoop [4], Spark [5], and Impala [6], are mainly optimized for offline analytics. They employ bulk loading at slack hours that brings a significant lag to contact recent data. Typical NoSQL systems, such as HBase [7] and Kudu [8], only provide single-row transactions that cannot be used in critical applications. Taking double-entry accounting in Fintech as an example, it needs to insert multiple related records transactionally. Many OLTP systems claim to meet this demand but may suffer from imperfect scalability [9].

To the best of our knowledge, there is still no silver bullet that could meet all demands for critical applications. Existing transaction models are either too strict (OLTP) or too relaxed (NoSQL) to make a trade-off among timeliness, scalability, and serializability. For this reason, we introduce a new approach called *transactional data ingestion* that helps OLAP systems to analyze historical and real-time data correctly. This model provides an append-only operation called *ingestion*, which could insert a small batch of records into target partitions transactionally. Meanwhile, for the upcoming OLAP query, it also ensures serializable isolation to scan data spanning multiple partitions, which would reflect both recently-ingested and historical data.

However, there is an issue when we implement the above operations with current transaction mechanisms. In distributed databases, *two-phase commits* (2PC) protocol (or its variants) is used to handle transactions spanning multiple partitions [10]. Under serializable isolation, 2PC will decide the serializable order and block conflicting partitions when different transactions access overlapped partitions. Here is a contradiction to optimize 2PC that requires costly coordination [11]. Intuitively, a transactional operation (query or ingestion) is "partial" when it only accesses one partition or is "distributed" when it needs to access multiple partitions. An analytical query would access multiple partitions to exploit distributed parallelism, while data ingestion prefers partial cases that could reduce potential conflicts. Thus, it is hard to optimize transactional ingestion within OLAP systems. Even though snapshot isolation enables non-blocking execution, it relaxes serializable constraints, which may be infeasible in critical scenarios.

To optimize both data ingestion and analytics with serializable isolation, we introduce a non-blocking protocol, called *metadata-oriented commit* (MOC). The basic idea of MOC is to detach transaction-related metadata from each partition. By narrowing the scope of metadata maintaining, this design brings two benefits. First, MOC turns each distributed transaction as multiple equivalent partial transactions to exploit the parallelism of multiple partitions. Second, to serve each

- *The authors are with the East China Normal University, Shanghai 200241, China. E-mail: {zhifangli, beichengpeng, qiulihuang}@stu.ecnu.edu.cn, clweng@dase.ecnu.edu.cn.*

*Manuscript received 26 Dec. 2018; revised 9 June 2020; accepted 13 July 2020.
Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Chuliang Weng.)
Recommended for acceptance by T. Palpanas.
Digital Object Identifier no. 10.1109/TKDE.2020.3011510*

TABLE 1
The Description of Main Terms

Term	Description
P_{all}	All partitions in the cluster
i_v, q_v	The ingestion/query with the version v
$P(i_v)$, $\|P(i_v)\|$	Accessed partitions of i_v, and the partition number
$B(i_v)$	Batch size of i_v
$L(i_v)$	The latency for the ingestion i_v
$TP(n)$	Ingestion throughput under n feeds
$D(n)$	The parallelism degree of ingestion with n feeds
MRPS	The number of metadata requests per second

upcoming query, it could spawn a lightweight "snapshot" from transaction-related metadata, which helps to scan consistent data from ingested partitions.

Besides, data ingestion also faces a challenge in scalability as an analytical system may run plenty of partitions to parallelize query execution. Consequently, when we need to ingest massive records into many partitions, data persistence, logging mechanisms, and data traffic become additional bottlenecks. With the advance of in-memory and network technologies, we employ several straightforward and effective ways to cope with data ingestion at scale.

The contributions of this paper are concluded as follows.

- We outline the real-time analytics workload with ingestion and show that the blocking 2PC of distributed transactions is the main bottleneck. Then we propose a protocol, namely MOC, which detaches transaction-related metadata from each partition.
- Based on MOC, we turn each distributed ingestion into equivalent partial operations. To ensure isolation for query execution, we also employ the lightweight snapshot based on transaction-related metadata.
- To cope with massive data, we introduce lazy persistence along with write-metadata logging. Besides, we also employ I/O multiplexing and locality-aware traffic to scale with many partition to ingest.
- We integrate the above mechanisms into our prototype system called Karst and then analyze it with relevant systems under a simulated ingestion workload. Experiments show up to about 2x$\sim$10x performance along with remarkable scalability.

Rest of paper is organized as follows. Section 2 outlines the background of data ingestion. Section 3 presents the design principle of MOC. Sections 4 and 5 introduce implementation details and optimization mechanisms. Section 6 experimentally evaluates Karst's performance and the impact of optimizations. Section 7 discusses related works. Finally, Section 8 concludes this paper.

2 Background

This section introduces the background of transactional data ingestion and lists main terms in Table 1.

2.1 Workload Definition

The workload of real-time analytics employs an *ingestion engine* and a *query engine* on the cluster shown in Fig. 1. The external world constructs the latest data into small batches and then pushes each generated batch to the log-based data feeds, such as Apache Kafka [12] and ActiveMQ [13]. The ingestion engine continuously pulls each cached batch from the connected feeds, and inserts contained records into the target partitions by a transactional ingestion. We denote the set of all partitions in the cluster by P_{all} for short. To execute an analytical SQL, the query engine scans P_{all} to aggregate both historical and ingested data into the final results.

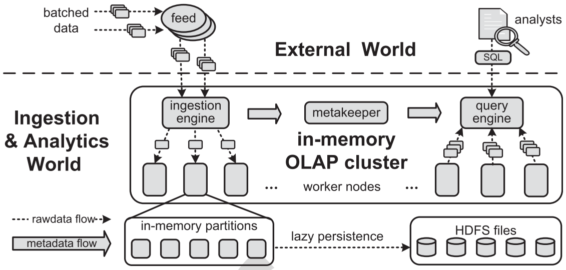


Fig. 1. The ingestion & analytics workload.

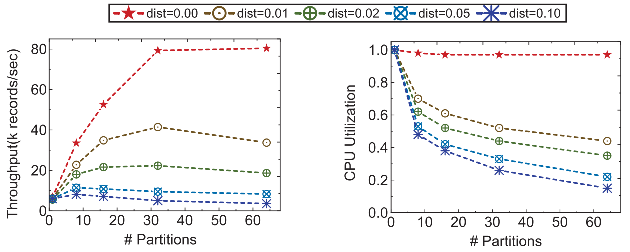
To outline this workload formally, we indicate i_v as the transactional ingestion to insert a batch of records, where v is the monotonically increasing version of P_{all} after i_v is committed. For the ingestion i_v, the set of ingested partitions is denoted as $P(i_v)$ with $|P(i_v)|$ partitions, and $|P(i_v)| \leq |P_{all}|$. According to the partitioning method (e.g., hash or range), all records in the batch of i_v should be inserted to each corresponding partition in $P(i_v)$. After all records are inserted successfully, i_v is committed to update the state of P_{all} and makes ingested data visible. If it fails, i_v is aborted to revoke ingested data. In most of analytical cases, data is append-only due to the format optimized for scanning. Consequently, OLAP systems usually do not support occasional in-place update/delete but turn it into out-of-place inserts on the original table and a delta table that indicates the deleted records [14], [15]. The delta table helps the query to filter invalid records from the original table. Therefore, changes could also be represented by equivalent ingestions before generating each batch for ingestion.

Similarly, each query q_v needs to scan $|P(q_v)|$ partitions in the $P(q_v)$. To gain correct and timely results, q_v should reflect recent data after all ingestions with the version prior to v are committed. To help understand this data dependency, we use "$\prec$" to indicate the partial order between the query q_v and previous ingestions as follows:

$$\{\ i_k \mid _{k \leq v}\} \prec q_v. \tag{1}$$

Under strict serializable isolation, q_v needs to wait for all pending ingestions $\{i_k|_{k \leq v}\}$ and scans their ingested data.

To accelerate data scanning, the query engine could employ in-memory storage, which is common in modern OLAP systems [15]. As memory is volatile, ingested data still needs to be persisted to durable storage finally. A feasible optimization is to lazily persist memory data in bulk to amortize the cost of disk write. However, this method faces the risk of data loss when node failures occur before data is persisted. *Write-ahead logging* (WAL) [16] is a popular technique to recover unpersisted data but could also involve significant overhead to log massive data. How to persist data safely with low cost is an additional challenge.



(a) Ingestion Throughput.

(b) Resource Utilization.

Fig. 2. The impact of access distribution.

2.2 Revisit Distributed Transaction for Data Ingestion

In this workload, ingestion and query are usually implemented by distributed transactions that need to cope with two types of potential conflicts in 2PC. (1) For two ingestions i_{v_1} and i_{v_2}, write-write conflicts occur when written partitions are overlapped (i.e., $P(i_{v_1}) \cap P(i_{v_2}) \neq \phi$). (2) For a query q_v, write-read conflicts occur when the scanned partitions are modified by other ingestions. The impact of conflicts is affected by how accessed data distributes among partitions [11]. To optimize the distribution transactions for ingestion or query, there are two feasible partitioning strategies that lead to different access distributions.

- The *txn-optimized* partitioning reduces potential conflicts by imposing each ingestion to run on a single partition (i.e., $|P(i_v)| \rightarrow 1$). At a cost, it may harm the parallelization of analytical query.
- The *query-optimized* partitioning optimizes analytics performance by parallelizing data scanning on many partitions (i.e., $|P(i_v)| \rightarrow |P_{all}|$). However, it also increases the conflicts of overlapped partitions.

Intuitively, when each operation only manipulates a single partition, different partitions can work simultaneously with excellent scalability. However, under query-optimized partitioning, the situation is the opposite [11]. To ensure serializable isolation, it will involve a blocking 2PC on the conflicting partitions. To resolve this issue, popular protocols, such as deterministic, timestamp-based, optimistic, or locking [17], mainly consider OLTP-like cases and need to maintain fine-grained metadata, such as pre-known write/read sets, write/read timestamps or locks for each individual record. They are hard to optimize analytical workloads, where write/read sets spanning multiple partitions are very large and ad-hoc. In essence, these methods assume that each operation is short-lived and operates a few of records, while write-heavy ingestion or read-heavy OLAP query would take a long execution time.

To verify the issue of 2PC, we evaluate data ingestion by using VoltDB [18], a partitioned database for real-time and serializable analysis via blocking 2PC. In experiments, stock records are ingested into the database under the varying partition number. We simulate the different access distributions by changing the value of $\frac{|P(i_v)|-1}{|P_{all}|}$ in each ingested batch (i.e., dist = 0.0 to 0.1). Fig. 2a shows that ingestion throughput degrades with the varying access distribution as more accessed partitions could increase the ratio of potential conflicts. Meanwhile, we use Linux Perf to collect CPU utilization during data ingestion. Experimental results in Fig. 2b show CPU idle caused by conflicts. We also observe that the increase of partitions makes the situation worse due to the longer ingestion time.

The above experiments show that data ingestion still faces performance issues under the query-optimized strategy. As partitioning selection is exclusive, it is hard to optimize both ingestion and OLAP query. The user has to choose one or the other strategy. In contrast, this paper aims to optimize data ingestion without breaking the query optimized data layout used in existing OLAP systems.

3 Metadata-Oriented Commit

Since the above analysis shows the shortcoming of blocking 2PC, this section introduces non-blocking *metadata-oriented commit* (MOC) for the ingestion and analytics workload.

3.1 Region-Based Metadata

In the OLTP systems, transaction-related metadata, including timestamps and locks, is maintained with each record [19]. Under query-optimized partitioning, 2PC needs to update metadata spanning multiple partitions atomically. Furthermore, the record-coupled method is too fine-grained for the ingestion and analytics workload that manipulates massive data within each operation.

The core idea of MOC is to reorganize transaction-related metadata. As both ingestion and analytical query manipulate contiguous records, it makes sense to employ a coarse-grained approach. Specifically, we regard each group of contiguous records as a *region* indicates by a vector < `Pt`, `Beg`, `Rag`, `Ver` > . In this vector, `Pt` is which partition of this region resides, `Beg` and `Rag` indicate the beginning position and range of records within this region. `Ver` is its commit version when data in the region is committed. Due to the append-only ingestion, regions can be allocated to the adjacent positions. Thus, transaction-related metadata with further compaction is small enough to be handled by a single node called *metakeeper*. As a consequence, expensive 2PC spanning multiple partitions can be replaced by a lightweight partial operation within the metakeeper.

Intuitively, each ingestion or query is independent when there are no others that access overlapped regions. By assigning each ingestion with disjoint regions, the metakeeper can resolve write-write conflicts in advance. To reduce the footprint of metadata, adjacent regions can be coalesced into a larger region. For instance, two regions $< p, 3, 2, v_1 >$ and $< p, 5, 3, v_2 >$ can be coalesced into a single $< p, 3, 5, max\{v_1, v_2\} >$. When assigned regions trend to be adjacent, fully coalesced region vectors only occupy a small amount of memory. This feature also helps to resolve write-read conflicts during query execution. For a query q_v, the metakeeper spawns a *snapshot* by coalescing region vectors that own the version prior to v. As this snapshot abstracts a consistent and up-to-date database state for q_v, write-read conflicts can be prevented by scanning data in the partitions according to the snapshot.

3.2 Non-Blocking Commit Protocol

With region-based metadata, we further introduce the non-blocking protocol of MOC. For each distributed operation, MOC follows three steps: (1) *request* metadata by contacting

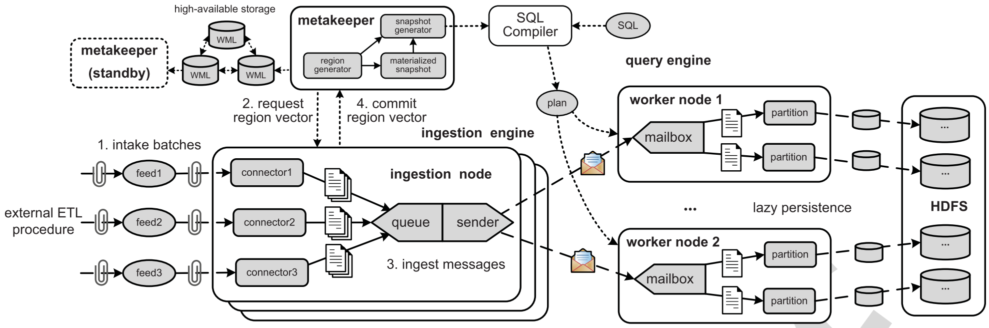


Fig. 3. The architecture of karst.

with the metakeeper, (2) *manipulate* target partitions in parallel, (3) *commit* metadata to the metakeeper.

Note that ingestion and query have a few differences. In the first step, each ingestion i_v requests region vectors for ingested partitions $P(i_v)$, while each query q_v requests a snapshot for scanned partitions $P(q_v)$. In the second step, i_v writes data into $P(i_v)$, while q_v scans data from $P(q_v)$ by its snapshot. Finally, i_v commits region vectors to the metakeeper, while read-only q_v could skip the commit step.

Since each step only accesses the metakeeper and the target partitions, this protocol equivalently converts each distributed transaction for ingestion or query into multiple partial operations via the above three steps. Specifically, each ingestion is turned into $1+|P(i_v)|+1$ partial operations and each query is turned into $1+|P(q_v)|$ partial operations. As each partial operation can be executed on the single partition without blocking, it exploits the parallelism of multiple partitions for both ingestion and query.

3.3 Performance Analysis

This subsection analyzes the performance of MOC. Ingestion throughput can be measured by the amount of inserted records per second. For n external feeds, due to potential conflicts, the real parallelism degree $D(n)$ may be less than n. Ideally, the ingestion throughput $TP(n)$ and the latency $L(i_v)$ to ingest each batch are given by:

$$TP(n) = min\{D(n)\cdot\frac{B(i_v)}{L(i_v)},\ F_b,\ MRPS\cdot B(i_v)\} \ |\ \ D(n) \le n$$
$$0 \ < \ L(i_v) \ \le \ 2\cdot|P(i_v)|\cdot L_p{+}L_m, \tag{2}$$

where $B(i_v)$ is the batch size of ingestion. As i_v contacts with $|P(i_v)|$ partitions twice (send data and receive ACKs), $L(i_v)$ has an upper bound of $2\cdot|P(i_v)|\cdot L_p{+}L_m$, where L_p is the elapsed time to manipulate data on each partition and L_m is the elapsed time to contact with metakeeper. In addition, $TP(n)$ is also limited by both the physical bandwidth F_b affected by hardware and metadata requests per second (MRPS), which is bounded by the metakeeper.

As each batch with hundreds of records to ingest only involves a lightweight metadata operation, the MOC protocol reduces the pressure on the metakeeper significantly. Even though the metakeeper is designed as a centralized structure to avoid costly 2PC, MRPS would not be the bottlenecks before F_b reaches its bound.

Formula 1. *To handle n external feeds, the ingestion throughput of 2PC and MOC is given by:*

$$TP_{2pc}(n) = D_{2pc}(n)\cdot\frac{B(i_v)}{L(i_v)} = min\{\lfloor\frac{|P_{all}|}{|P(i_v)|}\rfloor, n\}\cdot\frac{B(i_v)}{L(i_v)}$$
$$TP_{moc}(n) = D_{moc}(n)\cdot\frac{B(i_v)}{L(i_v)} = n\cdot\frac{B(i_v)}{L(i_v)}. \tag{3}$$

Proof. When F_b and MRPS are not reached, throughput $TP(n)$ is determined by the parallelism degree $D(n)$. By avoiding the blocking caused by coupled-metadata, MOC could run n ingestions simultaneously, i.e., $D_{moc}(n){=}n$. In contrast, 2PC cannot support more than $\lfloor\frac{|P_{all}|}{|P(i_v)|}\rfloor$ concurrent ingestions, i.e., $D_{2pc}(n){=}min\{\lfloor\frac{|P_{all}|}{|P(i_v)|}\rfloor,\ n\}$. □

From the above analysis, we observe that two facts. First, $TP_{2pc}(n)$ depends on the partitioning of the dataset inherently. As for txn-optimized partitioning (i.e., $|P(i_v)|\rightarrow 1$), 2PC shows a similar throughput like MOC but harms query performance for its suboptimal data distribution. As for query-optimized partitioning (i.e., $|P(i_v)|\rightarrow|P_{all}|$), 2PC shows merely $1/n$ throughput. Second, under query-optimized partitioning, it is necessary to optimize $L(i_v)$ when the batch size or partition number is large.

4 Implementation

To demonstrate how to integrate transactional data ingestion, we build our prototype system Karst based on Ginkgo [20], [21]. Initially, Ginkgo is an offline OLAP system upon an in-memory cluster and employs HDFS as its backup storage. Karst is a variant of Ginkgo that addresses data ingestion for real-time analytics. This section introduces the implementation of Karst, which can also inspire similar cases in other systems.

4.1 Main Components

The architecture of Karst is shown in Fig. 3 that consists of three main parts as follows.

The core of Karst is the metakeeper that assigns the ordered version and region-based metadata. The metakeeper provides two services based on MOC: (1) the ingestion service to generate region vectors for each ingestion request. (2) the query service to extract a snapshot for each query request. The metakeeper implements the server side and the client side based on an RPC library CAF [22]. To ensure transactional operations, all allocated region vectors and their commit information should be logged. In the basic setting, the metakeeper logs metadata changes in the file system. To enable high availability, the single point of failures can be handled by leveraging high-available storage, like a Redis cluster. The standby metakeeper can easily recover the services based on logged metadata.

The ingestion engine is responsible for pushing data into the target partitions. Before data ingestion, the external ETL procedure should construct incremental data into each batch and pushes prepared batches to the feed for caching. Each feed is handled by a *connector* running on the ingestion node of ingestion engine, which continuously pulls each cached batch from the connected feed, and inserts contained data to the target partitions. The setting of connectors and ingestion nodes depends on the number of external feeds.

Like typical OLAP systems, the partitions of dataset are held by one or more worker nodes to scale with computing resources. To execute an analytical SQL, the query engine needs to scan and process data from each accessed partition in parallel. Meanwhile, data is ingested into the partitions on the worker nodes. Each worker node employs memory as its primary storage to accelerate both ingestion and query. Besides, HDFS is used to persist ingested data for backup.

4.2 Ingestion Execution

To make a more detailed description, we conclude data ingestion execution as the following steps.

Intake Batches. Before ingestion, the external ETL continues to generate the sequence of batches based on the given size. During this step, incremental data on the original table (and on the delta table for occasional update/delete) is constructed as each batch before ingestion. Prepared batches are pushed to the target feed subsequently. To start an ingestion, the connector pulls a batch of data from the connected feed. Then the connector divides contained records according to the target partition of each batch.

Request Metadata. By using the client of metakeeper, the connector requests a monotonically increasing version v along with $|P(i_v)|$ non-overlapped region vectors on the ingested partitions $P(i_v)$. For each partition $p \in P(i_v)$ with n records to ingest, the metakeeper allocates a region in the form of $< p, b, n, v >$ and moves forward the beginning positions of the next regions. Then allocated region vectors are returned to the connector. The metakeeper maintains committed region vectors as transaction-related metadata in memory and logs their changes before requests are returned.

Ingest Messages. For each partition in $P(i_v)$, the connector encapsulates ingested records into a message with the header of its region vector. After that, the connector caches $|P(i_v)|$ generated messages in a queue, which are sent to the mailboxes of target worker nodes subsequently. In each worker node, the mailbox unpacks received messages, and inserts contained records into the given positions according to the corresponding region vector. Finally, an ACK is replied to indicate that this message is handled successfully.

Commit Metadata. When $|P(i_v)|$ ACKs are received before the timeout, the ingestion node commits i_v to the metakeeper to make ingested records in the region vectors visible. Otherwise, i_v is aborted to rollback its modifications.

4.3 Query Execution

To support real-time analytics, we upgrade the original query engine to execute the query under data ingestion. The SQL compiler is modified to put a metadata snapshot into each query plan so that the scan operator could read recently-ingested records with the help of the snapshot. The workflow to execute a query is described as follows.

Compile Query Plan. To start an OLAP query, the query engine compiles the SQL clause into a plan tree and traverses the plan tree to get the set of scanned partitions.

Generate Snapshot. Then the query engine uses the metakeeper client to request the current version v of database status. According to the given isolation, the metakeeper scans committed region vectors on the scanned partitions $P(q_v)$ and coalesces these region vectors that meet version requirements into a lightweight snapshot.

Apply Snapshot. The SQL compiler puts this snapshot into the scan operators in the query plan, which are dispatched to every partition in $P(q_v)$. With the snapshot, each scan operator reads the local partition and passes invalid records to the next operators in the query plan.

By modifying the SQL compiler and the scan operator, the above steps could also be applied to other analytical systems, without breaking their already existing architectures.

4.4 Query Isolation

To allow the user to make a trade-off among correctness, timeliness, and query response time, Karst supports three isolation levels by controlling snapshot generation.

Serializable Isolation ensures fully serializable and up-to-date results for critical applications. The query q_v needs to scan data from all ingestions prior to v as follows:

$$\{\ i_k \mid_{\ k \,\leq\, v} \} \prec q_v. \tag{4}$$

Since some ingestions may still be running during snapshot generation, each scan operator needs to wait for them to insert data into the local partition. We denote the set of waited ingestions as W_v, which can be given by:

$$W_v = \{\ i_k \mid_{\ k \,\leq\, v \,\wedge\, i_k \ is\ running} \}. \tag{5}$$

To minimize the impact on query execution, the scan operator could first read historical data and recently-committed data before waiting for W_v. Through this approach, this waiting time tends to be overlapped by the scanning time, especially for long-runtime queries in the OLAP workloads.

Read Committed Isolation is a relaxed version when fully serializable results are not compulsive. It avoids this waiting time by skipping all uncommitted data in W_v. Under this isolation, q_v only requires the committed ingestions

$$\{\ i_k \mid_{\ k \,\leq\, v \,\wedge\, i_k \ is\ committed} \} \prec q_v. \tag{6}$$

To generate the snapshot for q_v, the metakeeper needs to coalesce all committed region vectors with the version equal or less than v. Due to discrete ingestions scanned by the query, it may break the serializable version sequence, but it is still acceptable for some non-critical applications.

Snapshot Isolation is similar to read committed isolation but further skips the ingestions with discrete versions. Under this isolation, q_v attempts to scan an earlier state of database as follows:

$$\{\ i_k \mid_{\ k \,\leq\, v^{*} \leq\, v \,\wedge\, i_k \ is\ committed} \} \prec q_v, \tag{7}$$

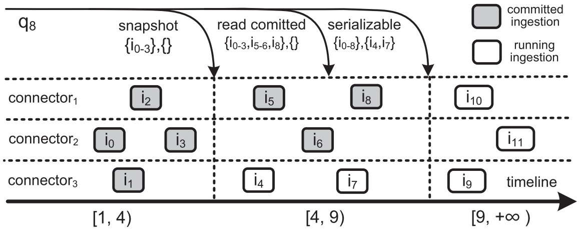


Fig. 4. The scanned and waited ingestions for q_8.

where discrete ingestions in $\{i_k|_{\ v^* <\ k \leq v}\}$ are skipped to ensure a serializable version sequence. To get the snapshot for q_v, the metakeeper tracks the maximal adjacent version v^* in committed region vectors. Then the metakeeper coalesces these committed region vectors that are prior to v^* as the snapshot. By default, Karst is configured as snapshot isolation, which could make a trade-off among correctness, timeliness, and query response time.

We take Fig. 4 as an example to illustrate different isolation levels. For the query q_8, the serializable isolation needs to read data from ingestions $\{i_{0\sim8}\}$ with versions prior to 8, in which data in $\{i_4, i_7\}$ should be waited by the scan operator. The snapshot isolation only reads $\{i_{0\sim3}\}$ without waiting for running ingestions, and the read committed isolation needs to further read extra data from $\{i_{5\sim6}, i_8\}$.

4.5 Metadata Compaction

Due to endless data ingestion, accumulated metadata will incur a huge memory footprint. To avoid this potential bottleneck of the centralized structure, the metakeeper employs two strategies to compact metadata.

Snapshot Materialization. When plenty of ingestions have been committed, it may take a long time to generate the snapshot by scanning these region vectors. To accelerate this step, the metakeeper periodically pre-coalesces region vectors as materialized snapshots and caches them in memory for subsequent query requests. As region vectors are allocated at the adjacent positions, materialized snapshots have a minor size. For a query q_v, the metakeeper could combine a recently materialized snapshot of $\{i_k|_{k\leq v^*}\}$ with a few region vectors of $\{i_k|_{v^* < k \leq v}\}$ into the final snapshot.

Garbage Collection. After region vectors have been materialized, they could be discarded as subsequent query execution will rely on the materialized snapshot. Besides, materialized snapshots could also be removed when their versions fall behind all running queries. Consequently, the metakeeper could remove both useless region vectors and materialized snapshots to compact metadata.

With these compaction methods, the footprint of metadata has a limited upper bound (a few MB in Section 6.2 experiments). Therefore, it is feasible to hold all transaction-related metadata within a centralized metakeeper to avoid blocking 2PC and will not lead to the bottleneck.

5 Optimizations

To ingest massive data into plenty of partitions, this section further optimizes data persistence, logging, and data traffic.

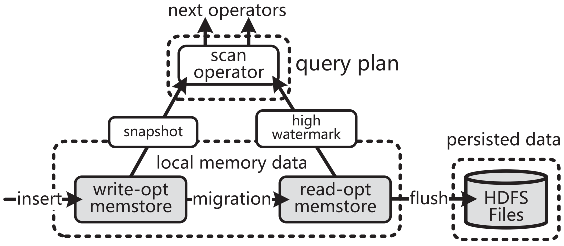


Fig. 5. The lazy persistence of each partition.

5.1 Lazy Persistence

Intuitively, to ingest each batch into $|P(i_v)|$ partitions, it is costly to immediately persist data before returning ACKs, especially when there are many partitions. For this reason, we carry out data persistence in a lazy manner. In the systems for in-memory analytics, each partition usually stores data in the *Read-opt Memstore* (RMS). To integrate data ingestion, we add an extra *Write-opt Memstore* (WMS) to hold recently-ingested data. Normally, WMS organizes data to optimize write, while RMS organizes data to optimize scanning. Each partition primarily inserts received records into WMS. To optimize for the subsequent OLAP query, ingested records are migrated to RMS asynchronously. Finally, due to volatile and space-limited memory, Karst flushes RMS data to durable and high-available HDFS files.

As lazy persistence faces a risk of data loss when worker nodes fail, we introduce a watermark-based mechanism to ensure data integrity. For each partition, the metakeeper maintains a *high watermark* for RMS data and a *low watermark* for persisted HDFS data. During lazy persistence, low watermarks chase after high watermarks, while high watermarks chase after allocated regions. Based on the two monotonically increasing watermarks, lazy persistence is carried out with the two following steps.

Migration. The high watermark indicates the maximum readable position of RMS data that has been migrated from WMS. For each partition, the metakeeper computes a new high watermark by coalescing the current watermark with adjacent committed region vectors. After Karst migrates incremental WMS data between the current and new high watermark to RMS, this new high watermark is committed to expand the readable region of RMS. Due to the increasing of RMS data, this step could optimize data scanning in the subsequent queries.

Flush. Similarly, the low watermark indicates the maximum position of persisted HDFS data that has been flushed from RMS. For each partition, the metakeeper leverages the current high watermark as the new low watermark and then flushes incremental RMS data between the current and the new low watermark to the HDFS file. After this step has been accomplished, this new low watermark is committed to the metakeeper. Otherwise, the dirty tail of HDFS file is truncated to the length of the current low watermark.

When lazy persistence is involved, the scan operator of query needs to read both RMS data and WMS data that exist underlying data migration, as shown in Fig. 5. Specifically, the scan operator reads RMS data below the high watermarks and reads WMS data with the snapshot that contains region vectors above the high watermarks. During persistence, the high watermarks are increased monotonously, and WMS data is preserved until all dependent queries are accomplished. Thus, the concurrent query will not be affected by lazy persistence.

5.2 Write-Metadata Logging

Under lazy persistence, even if new high watermarks are committed, unpersisted RMS data in memory is still volatile. In current databases, *Write-Ahead Logging* (WAL) ensures the durability of committed transactions by writing changed data to the file [16]. Unfortunately, WAL mainly considers OLTP scenarios like TPC-C or YCSB, where each transaction only writes a few records, while data ingestion needs to write many records. We notice that logging and data persistence write the same value twice. This redundant logging dramatically increases the overhead in data ingestion.

To avoid redundant logging in WAL, we further introduce *Write-Metadata Logging* (WML) that works with lazy persistence. Our idea comes from the fact that lost records can be retrieved from the common upstream feeds, such as Kafka, which could preserve cached data for a given time. Therefore, WML only needs to log lightweight metadata, including region vectors and watermarks, rather than the value of data. When worker nodes fail, standby connectors could fetch WML to seek failed ingestions and re-ingest data from the upstream feeds. Owing to the idempotent re-ingestion within the same region, recovery based on WML can ensure exactly-once semantics eventually. Note that the flush period in lazy persistence should be shorter than the data preserved time of upstream feeds.

In essence, the recovery time of WML depends on ingestion throughput, which may be longer than WAL that directly scans log files. As failure recovery occupies minor time during normal execution, heavyweight WAL may harm runtime performance unworthily. With lazy persistence that flushes memory data periodically, WML has a bounded recovery time. An experiment in Section 6.6 illustrates that WML significantly improves ingestion performance and only introduces an acceptable recovery overhead. From this point of view, we believe that WML outperforms WAL in the scenarios of data ingestion.

5.3 I/O Multiplexing

Traditional network programming leverages multi-threaded I/O that each thread handles a TCP connection. This classic model simplifies programming and works well with a few connections. Unfortunately, a scalable OLAP system could run plenty of worker nodes. When there are many connections between the ingestion node and worker nodes, CPU resources are wasted on context switch rather than ingestion execution, especially for the smaller batch sizes.

Recently, I/O multiplexing, serving multiple connections simultaneously, becomes a feasible method to cope with the above issue [23]. In Karst, both the ingestion node and the worker node run a listening thread based on *epoll* – Linux's I/O multiplexing API. Once a group of connections is ready, the listening thread is notified to send messages to the corresponding worker nodes or insert received data to the partitions by one shot. Experiments in Section 6.4 illustrate that I/O multiplexing scales better than multi-threaded I/O.

5.4 Locality-Aware Traffic

Since the growing cores in modern servers, scalable OLAP systems may also hold plenty of partitions to take advantage of parallelism. Under query-optimized partitioning (i.e., $|P(i_v)| \approx |P_{all}|$), the number of invoked syscalls for each ingestion is approximate to $2\times|P_{all}|$. For example, when 1,000 batches are ingested into 64 partitions, there are noticeable $2\times 64\times 1,000$ syscalls (i.e., 128,000).

To amortize the overhead of syscalls, we employ *locality-aware traffic* that is aware of the location of ingested partitions. During message generation, the connector divides each batch of records into messages based on the target worker nodes rather than the target partition. Through this method, data for multiple partitions within one worker node can be transferred via a single syscall. After the worker node inserts received data to corresponding partitions, an ACK is replied similarly. When each worker node contains 16 partitions, locality-aware traffic reduces the syscall number to a far less $2\times(64/16)\times 1000$ (i.e., $8,000 \ll 128,000$). Experiments in Section 6.4 show that locality-aware traffic improves scalability when each worker node holds the increasing number of partitions.

6 Evaluation

In this section, we evaluate the Karst performance through both micro-benchmark and macro-benchmark.

6.1 Experiments Setup

Our experiments use nine x86 servers, which are equipped with two E5-2630 CPUs with 20 cores, 128 GB memory, and four 1 TB 7200 RPM disks. All servers run 64-bit Centos 7, which are connected by the 10 Gbps Ethernet. This cluster deploys Hadoop 2.7 that could truncate the HDFS file to a given length. One server runs the ingestion node and the metakeeper, and the other eight servers run worker nodes to serve query execution.

To simulate the workload of real-time ingestion, we leverage a real financial dataset of the *Shanghai Stock Exchange* (SSE) during September of 2010. The SSE dataset has a *trade* table with more than 50 GB records of stock trading (about 120 bytes per record). Besides, we also leverage a synthetic OLAP benchmark TPC-H with the scale factor at 100 GB (about 130 bytes per record), which is similar to SSE. The schema of SSE data is as follows:

trade (trade_no, trade_date, trade_time, trade_time_dec, order_time, order_time_dec, order_no, trade_price, trade_amt, trade_vol, sec_code, pbu_id, acct_id, trade_dir, order_prtfil_code, tran_type, trade_type, proc_type, order_type, stat_pbu_id).

6.2 Micro-Benchmark for the Metakeeper

Before evaluating overall performance, we first need to ensure that the centralized metakeeper is not the bottleneck for ingestion or query. Therefore, we employ a micro-benchmark to evaluate the metakeeper with 20 clients.

Simple Workload. In this experiment, we evaluate the "simple workload" that each client requests metadata for ingestion or query, respectively. We change the number of partitions in a request. Table 2 shows that the metakeeper can serve more than 100 k metadata requests for ingestion or 50 k requests for query per second (i.e., MRPS). In most

TABLE 2
The Simple Workload

Types	Metrics	# Partitions						
		1	2	4	8	16	32	64
ingestion	MRPS (k)	203	201	196	185	159	128	105
	L_m (ms)	0.17	0.17	0.18	0.19	0.22	0.27	0.34
	$B(i_v)$·MRPS (GB/s)	36.3	35.9	35.1	33.1	28.4	22.9	18.8
query	MRPS (k)	238	220	202	172	123	88	65
	L_m (ms)	0.16	0.17	0.18	0.21	0.28	0.33	0.58

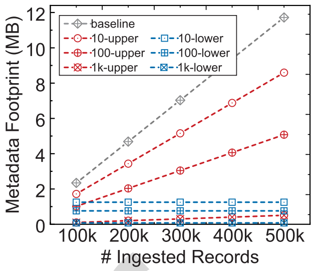


Fig. 7. Metadata footprint.

cases, each request is handled within 0.5 ms – far less than the whole runtime of ingestion or query. In ideal conditions, when the batch size is 1,600 records (about 150 KB), the metakeeper could serve at 30 GB/s throughput, which is much larger than the bandwidth of Gigabit Ethernet. By using a larger batch size, this value can further increase.

Mixed Workload. In this experiment, we evaluate the mixed workload that the metakeeper serves query requests under concurrent ingestion requests. We use fixed eight partitions in each request. Fig. 6 shows the latency distribution of query requests. Under ingestion requests (10 to 10 k MRPS), a query request could be handled within 2.0 ms (i.e., L_m). This low latency is mainly due to region-based and compacted metadata that amortizes the cost to get a lightweight snapshot from the metakeeper. The next experiments verify this viewpoint by diving into metakeeper.

Metadata Footprint. Fig. 7 shows how metadata footprint grows with concurrent ingestion requests. Each ingestion request will generate region vectors for 64 partitions. The baseline in the figure indicates the classic method that encapsulates transaction-related metadata along with each record. With records are ingested, the metadata footprint shows a linear growth. Then we employ region-based metadata and change the batch size "n" from 10 to 1 k records in each ingestion request. The upper bound of metadata footprint (n-upper) has a slower growing trend than the baseline. With compaction enabled, region vectors are materialized as lightweight snapshots. Therefore, the footprint shrinks to the lower bound (n-lower) of a few MB. We also notice that the larger batch size causes a smaller footprint as more records in the batch can share a single region vector.

Snapshot Size. Fig. 8 illustrates how the snapshot size grows with the partition number. By taking advantage of adjacent region allocation, committed regions could be coalesced into larger ones. The snapshot size is between an upper bound (uncoalesced) and a lower bound (fully coalesced) that only occupies a few KB size. Thus, getting a lightweight snapshot merely involves minor overhead for a long-running OLAP query.

These micro-benchmark experiments have demonstrated that the centralized metakeeper is sufficient for intensive ingestion or query requests of region-based metadata.

6.3 Macro-Benchmark for Data Ingestion

This subsection evaluates the performance of data ingestion via a macro-benchmark. In the whole lag to analyze data, the time to go through the external feed is affected by the implementation feed. For example, Kafka could introduce about 5 ~ 140 ms delay when the batch size of data increases from 100 to 3,200 records. Thus, to avoid the influence of external feed and focus on ingestion itself, we build a *bench-connector* that could pre-load data and then carry out data ingestion. In each experiment, we increase the number of bench-connector and collect performance metrics until the evaluated system is saturated, where throughput cannot scale with more bench-connectors. To meet the case of real-time analytics, the ingested SSE table is partitioned by the query-optimized key (e.g., "trade_no") so that each ingestion or query tends to access every partition. A part of the dataset is pre-ingested to warm-up the system.

The Impact of Batch Size. As discussed in Section 3.3, for a fixed partition number, ingestion performance is mainly affected by the batch size rather than the ingested record value. This experiment evaluates how this factor impacts ingestion performance. We leverage the bench-connectors to ingest SSE data into Karst with eight partitions. Fig. 9 shows ingestion throughput and latency under the varying batch size. The smaller batch has a shorter ingestion latency but a lower throughput due to more frequent metadata requests and data transfer, while the larger batch is the opposite. In the "sweet area" of this figure, the batch size is about 1,600 records (about 150 KB), which could be a trade-off between ingestion throughput and latency.

Analysis of Relevant Systems. In this experiment, we also attempt to analyze other relevant systems that declare for real-time ingestion, rather than bulk loading. Since each

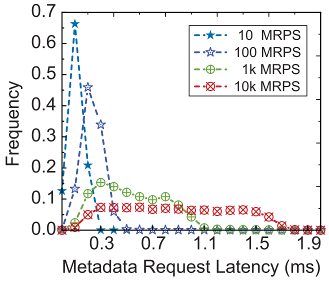


Fig. 6. Mixed workload.

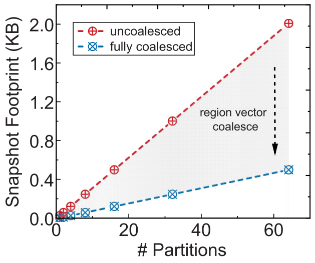


Fig. 8. Snapshot size.

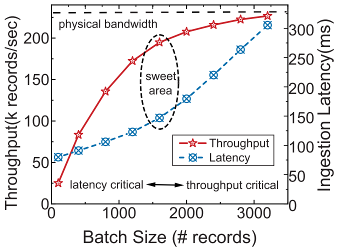


Fig. 9. Throughput versus latency.

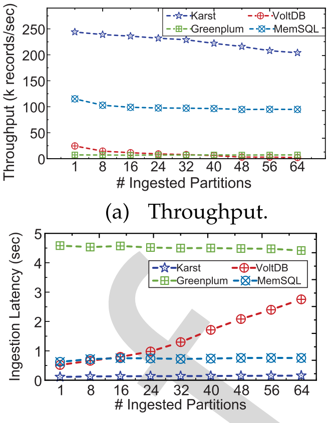
system may have a different transaction model, we leverage its built-in model to achieve transactional data ingestion. Besides, we also extend the bench-connector to adapt the transaction execution interface of each system. To make a fair comparison and also achieve the maximum performance, all systems do not employ replication mechanisms and leverage the local file system for logging. In this experiment, we use three types of batch sizes: small (400 records), medium (1,600 records), and large (3,200 records). Their throughput and latency are shown in Figs. 10, 11, and 12, respectively. For each system, we analyze its characteristics and application scenarios as follows.

Karst is equipped with non-blocking MOC to achieve the highest throughput of more than 200 k records per second under the medium batch size. Owing to optimized persistence and logging, Karst also achieves the ingestion latency of less than 200 ms. Compared with similar relational systems like VoltDB, Greenplum, and MemSQL, Karst has up to 2x∼10x throughput and also provides serializable isolation. More importantly, with I/O multiplexing and locality-aware traffic, Karst scales well with the growing partition number. Even though the small batch size has a lower throughput, it could achieve low latency less than 100 ms. We notice that the throughput gap between large and medium batch sizes is minor as the medium batch size has almost exploited the performance potentials.

VoltDB [18] is a partitioned database that stores data in an in-memory cluster, which is popular in finance analysis. VoltDB abstracts each transactional operation (modification or query) as a stored procedure and executes stored procedures under serializable isolation. To analyze VoltDB, the bench-connector runs pre-compiled stored procedures to insert a batch of records within each procedure invocation. We notice its inferior performance in this experiment, which is mainly caused by two reasons. (1) Its transaction mechanism is not optimized for data ingestion, where each batch would live a much longer time than general OLTP cases. (2) The query-optimized partitioning incurs blocking 2PC to resolve distributed transactions. VoltDB is saturated under only 2∼3 bench-connectors, while Karst could serve more than 20 bench-connectors. Even though VoltDB could achieve scalable data ingestion under txn-optimized partitioning, it limits complex analytics, including multi-table join.

Greenplum [24] and *MemSQL* [25] well support complex queries but employ relaxed snapshot isolation and read committed isolation. To analyze Greenplum and MemSQL, the bench-connector employs the standard JDBC interface to ingest each batch of records within a transaction. By relaxing isolation guarantee, Greenplum achieves scalable distributed transactions but still suffers from a significant ingestion latency caused by disk-based storage. MemSQL is similar to Greenplum but uses in-memory storage to achieve higher ingestion throughput and lower latency. However, owing to costly WAL in data ingestion, MemSQL still has lower performance than Karst, which employs lazy persistence and WML to reduce the overhead of writing data.

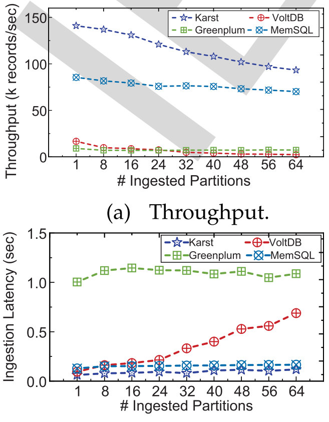
Apart from these relational systems, we also attempt to analyze other systems based on different transaction models that cannot be compared with the above systems directly.



(a) Throughput.

(b) Latency.

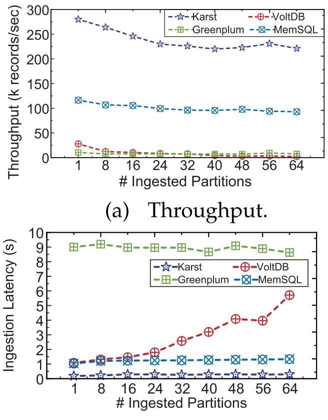
Fig. 11. Ingestion (medium batch).



(a) Throughput.

(b) Latency.

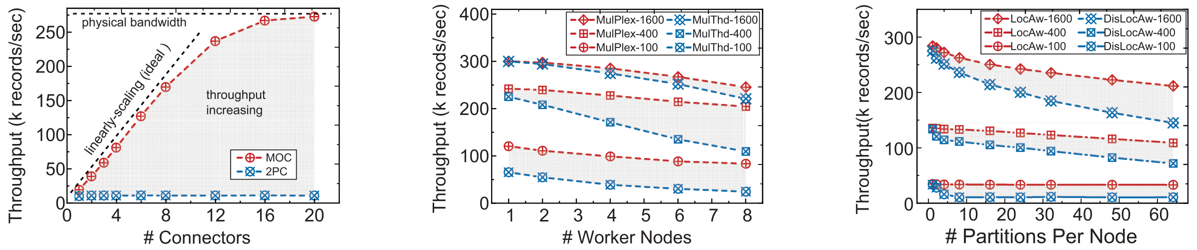
Fig. 10. Ingestion (small batch).



(a) Throughput.

(b) Latency.

Fig. 12. Ingestion (large batch).



(a) Connector Number Scaling. (b) Cluster Size Scaling. (c) Partition Density Scaling.

Fig. 13. Scalability validation.

Kudu [8] is a NoSQL-like system optimized for both write and read that declares for real-time analytics. Like typical NoSQL systems, Kudu only supports single-row transactions. To analyze Kudu, the bench-connector employs its API client to carry out data insert, which is configured with auto flush. Kudu shows less than 1-millisecond latency per record along with 25k $\sim$ 30k records per second throughput under the varying partition number. Even though it could achieve the lowest latency and acceptable throughput, its single-row transactions provide the weaker isolation that may not meet demands in critical applications.

Druid [26] is a real-time warehouse with txn-optimized partitioning. Specifically, Druid employs a special partition to hold all recently-ingested data so that ingestions always execute on this partition to avoid 2PC for distributed transactions. In experiments, the bench-connector uses its API client to ingest records, which shows about 20k $\sim$ 30k records per second throughput but is still worse than Karst for its HDFS storage. However, due to restricted partitioning, Druid has weaker join support in analytics.

6.4 Scalability Validation

In this experiment, we focus on the scalability of Karst from three aspects as follows.

Connector Number Scaling. First, we evaluate whether Karst based on MOC scales well with the growing connector number when it needs to deal with various feeds. We set the batch size as 1,600 records and increase the number of bench-connectors to ingest data into the Karst with eight partitions based on query-optimized partitioning. Fig. 13a shows that blocking 2PC in VoltDB has poor parallelism when more bench-connectors are involved. In contrast, as analyzed in Formula (1), non-blocking MOC has linear scalability before it is restricted by the physical bandwidth.

Cluster Size Scaling. Second, we evaluate whether Karst with I/O multiplexing scales well with the number of ingested worker nodes in the cluster. With different batch sizes (100 to 1,600 records), we employ different transfer methods to ingest data into the varying worker node number, where each worker node holds one partition. With the enlarged cluster scale, Fig. 13b shows that I/O multiplexing (i.e., MulPlex) outperforms multi-threaded I/O (i.e., MulThd) with up to $1.8\times\sim3.2\times$ speedup. Besides, I/O multiplexing also shows a slighter degradation when more worker nodes are involved, especially for smaller batch sizes. This is because of its much less context switch for transfer.

Scaling With Partition Density. Finally, we evaluate whether Karst scales well with the partition density when the worker node holds many partitions to be ingested. Under the different batch sizes (100 to 1,600 records), we change the number of partitions within each worker node. Fig. 13c illustrates the performance of data ingestion when locality-aware traffic is enabled (i.e., LocAw) or disabled (i.e., DisLocAw). With the enlarged partition density, locality-aware traffic shows up to $30\%\sim65\%$ promotion. By amortizing the overhead of invocations in transfer, it alleviates throughput degradation when the partition density increases. We also observe that this alleviation is more significant for smaller batch sizes that lead to more frequent invocations.

6.5 The Impact on Query Performance

In this experiment, we evaluate how concurrent ingestions impact on query execution. We choose three SSE SQLs for trade recommendations (see Table 3) and five TPC-H SQLs for business analytics. Before the experiment, we preload the whole datasets into Karst and set them as eight partitions by the query-optimized keys (e.g., “trade_no” in SSE's trade and “l_orderkey” in TPC-H's lineitem). Without concurrent ingestions, we execute above SQLs to get the baseline query execution time, as shown in Table 3. Due to multi-table joins, TPC-H SQLs spend a much longer time than SSE SQLs.

After that, we run three bench-connectors to ingest data and repeat each SQL to get the lag of query execution. We

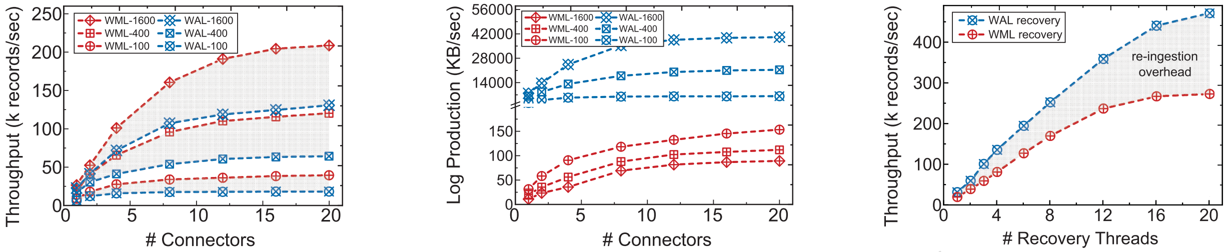
TABLE 3
Query Execution Time (Second)

	SSE SQLs			TPC-H SQLs				
	SSE Q1	SSE Q2	SSE Q3	TPC-H Q1	TPC-H Q3	TPC-H Q5	TPC-H Q10	TPC-H Q12
Baseline	1.16	0.87	1.09	12.18	156.04	58.97	172.53	9.52
ReadCommitted	1.24 (+6.52%)	0.93 (+6.34%)	1.16 (+6.72%)	12.82 (+5.29%)	162.81 (+4.34%)	61.96 (+5.06%)	179.83 (+4.23%)	10.10 (+6.10%)
Snapshot	1.22 (+5.31%)	0.92 (+5.38%)	1.15 (+6.02%)	12.77 (+4.87%)	162.45 (+4.11%)	61.99 (+5.13%)	179.40 (+3.98%)	10.07 (+5.80%)
Serializable	1.27 (+9.85%)	0.95 (+9.54%)	1.20 (+10.14%)	12.89 (+5.85%)	163.26 (+4.63%)	62.25 (+5.56%)	180.17 (+4.43%)	10.24 (+7.53%)

SSE Q1: select avg(trade_price), trade_date from trade where sec_code = XXXX and trade_date = ’2010-09-16’ and trade_time between ’10:40:00’ and ’11:00:00’ group by trade_date order by trade_date (the price trend of a stock).

SSE Q2: select max(trade_price), min(trade_price) from trade where sec_code = XXXX and trade_date = ’2010-09-16’ and trade_time between ’10:40:00’ and ’11:00:00’ (the maximal and minimal prices of a stock).

SSE Q3: select trade_vol, count(*) as fq from trade where sec_code = XXXX and trade_date = ’2010-09-16’ and trade_time between ’10:40:00’ and ’11:00:00’ group by trade_vol order by fq desc (the trading distribution of a stock).



(a) Ingestion Throughput. (b) Log Production. (c) Recovery Throughput.

Fig. 14. The impact of logging mechanism.

configure three isolation levels of Karst as discussed in Section 4.4. Under snapshot isolation, the query execution time has an acceptable 4%∼7% increase. The read committed isolation causes a slightly longer time as the query would scan more ingested data. Under serializable isolation, SSE SQLs show about 9.5%∼10% degradation due to waiting for a few uncommitted ingestions. Besides, we observe that the gap between different isolation levels are smaller in TPC-H SQLs because the waiting time could be overlapped during the long-running query execution.

6.6 The Impact of Logging Mechanism

In this experiment, we evaluate how different logging mechanisms impact on data ingestion. By using WAL and WML, we first carry out data ingestion with the varying batch size (100 to 1,600 records) and fixed eight partitions. We collect the ingestion throughput and the amount of log production during experiments. In Fig. 14a, WML shows 1.2×∼2.1× throughput over WAL. The phenomenon is caused by the different amounts of costly disk write. As shown in Fig. 14b, WAL produces more than two orders of magnitude log data than WML by skipping write data value. The relative improvements of WML are more significant for the smaller batch sizes that cause frequent logging. Besides, we also evaluate the recovery throughput with different thread numbers. Fig. 14c shows that WML only involves an acceptable 30%∼40% degradation of recovery throughput. This extra overhead is caused by the re-processing of unpersisted data. Even though WAL owns better recovery performance, WML outperforms WAL during normal data ingestion, which dominates the majority of execution time.

7 Related Work

As data ingestion for real-time analytics is still an open issue, we briefly introduce the relevant works.

Bulk Loading. Gobblin [27], Sqoop [28], Gpfdist [24], InstantLoading [29] and Flume [30] mainly consider bulk loading in OLAP databases. Owing to significant lags, bulk loading usually runs at slack hours. Furthermore, as they are external plugins, the query view of internal databases is not promised to be timely and correct during loading time. Mesa [31] periodically runs MapReduce procedures to load files every few minutes, while Karst runs continuous data ingestion with sub-second latency.

Streaming. Storm [32], Flink [33], Spark Streaming [34] or even streaming/batch combined Lambda systems [35] are mainly designed for on-line processing. They launch the continuous query to filter arrival data, while real-time analytics needs to scan both historical and ingested data in an ad-hoc manner. In fact, streaming systems normally play as the front-end ETL tool to prepare data before ingestion.

HTAP. Today, *Hybrid OLTP/OLAP* (HTAP) that performs query and update on a unified dataset is rising rapidly. Many NoSQL-based HTAP systems, including HBase [7], Kudu [8], Wildfile [36], VectorH [37], Es2[38], SnappyData [39], support transactional single-row insert along with OLAP query. Relational HTAP systems, such as Greenplum [24], MemSQL [25], Delta [40], mainly provide read committed or snapshot isolation (or called versioning) for OLAP query execution. Janus [41] joints an OLTP system and an OLAP system with a data pipeline but still employs 2PC to migrate data in bulk. Databus [42] studies how to capture data changes from existing OLTP systems but does not focus on how to ingest data to OLAP systems. Besides, centralized HTAP systems like HANA [15], Hyper [43], SQLServer[44] support full OLTP features in a highly-optimized node. BatchDB [45] and ScyPer [46] separate OLTP and OLAP to two replications to avoid conflicts. As they are mainly based on replication rather than partition, they have limited scalability compared with partitioned fashion. This paper does not build a new HTAP system but attempts to integrate existing partitioned OLAP systems with transactional data ingestion based on MOC.

Distributed Transactions. Many works try to optimize distributed transactions through special protocols or hardware [11], [17], [47], [48], [49], [50], [51]. However, they mainly consider typical OLTP workloads and are hard to apply for ingestion due to two reasons. (1) They assume that each operation is transient and frequent, while data ingestion and analytical query are long-running. (2) They consider general data dependency, while Karst is aware of append-only ingestion and read-only query. ParallelRaft [52] is semantic-aware to reduce unnecessary blocking but mainly considers replicated OLTP cases, while this paper focuses on partitioned OLAP cases. Druid [26] and Pinot [53] employ txn-optimized partitioning to avoid blocking 2PC. As a compromise, they poorly support analytics with joins, while Karst does not break existing query-optimized partitioning.

Transaction-Related Metadata. The SQL-on-Hadoop system HAWQ [54] stores metadata in a decoupled database to simplify distributed transactions. Owing to restricted HDFS, HAWQ has poor transaction performance and does not provide real serializable isolation. Megastore [55] and ElasTraS [56] also employ decoupled metadata but with record-based granularity. Cubrick [57] leverages coarse-grained but coupled metadata, which still involves 2PC in distributed transactions. Karst leverages region-based and decoupled metadata to optimize append-only ingestions and read-only queries without the blocking of distributed commit.

8 Conclusion

This paper addresses a subtle issue that, for real-time and critical analytics, transactional ingestion and OLAP query seem to be incompatible. Through our analysis and experiments, we demonstrate that blocking 2PC is the main bottleneck. By employing the metadata-oriented commit, Karst converts each distributed transaction into multiple partial operations to avoid 2PC. To ingest massive data into plenty of partitions, Karst also optimizes data persistence, logging mechanism, and data traffic. We integrate Karst into an existing OLAP system, which shows noticeable performance and scalability in data ingestion. This paper explores a scalable architecture for transactional data ingestion and also encourages future works on real-time analytics.

Acknowledgments

This research was supported by National Key Research and Development Program of China (No. 2018YFB1003400), and National Natural Science Foundation of China (No. 61772204 and 61732014).

References

[1] Case study: Bank deploys real-time analytics to improve trade recommendations. [Online]. Available: https://www.voltdb.com/files/case-study-finserv-bank/

[2] Technical overview: Voltdb for financial services. [Online]. Available: https://www.voltdb.com/files/technical-overview-voltdb-financial-services/

[3] A. Pavlo and M. Aslett, "What's really new with NewSQL?" *ACM SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, 2016.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 10.

[6] M. Bittorf *et al.*, "Impala: A modern, open-source SQL engine for hadoop," in *Proc. Conf. Innovative Database Res.*, 2015.

[7] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[8] T. Lipcon *et al.*, "Kudu: Storage for fast analytics on fast data." [Online]. Available: https://kudu.apache.org/kudu.pdf.

[9] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1771–1775.

[10] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 1150–1160.

[11] A. Pavlo, C. Curino, and S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 61–72.

[12] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. Int. Workshop Netw. Meets Databases*, 2011, pp. 1–7.

[13] [Online]. Available: http://activemq.apache.org/

[14] M. Stonebraker *et al.*, "C-store: A column-oriented DBMS," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 553–564.

[15] V. Sikka, F. Färber, A. Goel, and W. Lehner, "SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform," *Proc. VLDB Endowment*, vol. 6, pp. 1184–1185, 2013.

[16] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking main memory OLTP recovery," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 604–615.

[17] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endowment*, vol. 10, pp. 553–564, 2017.

[18] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data(base) Eng. Bulletin*, vol. 36, no. 2, pp. 21–27, 2013.

[19] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in PostgreSQL," *Proc. VLDB Endowment*, vol. 5, pp. 1850–1861, 2012.

[20] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton, "Elastic pipelining in an in-memory database cluster," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1279–1294.

[21] Z. Fang, C. Weng, L. Wang, H. Hu, and A. Zhou, "Scheduling resources to multiple pipelines of one query in a main memory database cluster," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 3, pp. 533–546, Mar. 2020.

[22] [Online]. Available: https://actor-framework.org/

[23] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming*, vol. 1. Reading, MA, USA: Addison-Wesley, 2004.

[24] [Online]. Available: https://pivotal.io/pivotal-greenplum.

[25] [Online]. Available: https://www.memsql.com/.

[26] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 157–168.

[27] L. Qiao *et al.*, "Gobblin: Unifying data ingestion for hadoop," *Proc. VLDB Endowment*, vol. 8, pp. 1764–1769, 2015.

[28] [Online]. Available: http://sqoop.apache.org/

[29] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *Proc. VLDB Endowment*, vol. 6, pp. 1702–1713, 2013.

[30] [Online]. Available: http://flume.apache.org/

[31] A. Gupta *et al.*, "Mesa: Geo-replicated, near real-time, scalable data warehousing," *Proc. VLDB Endowment*, vol. 7, pp. 1259–1270, 2014.

[32] A. Toshniwal *et al.*, "Storm@ twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.

[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink$^{\text{TM}}$: Stream and batch processing in a single engine," *IEEE Trans. Knowl. Data Eng.*, vol. 38, no. 4, pp. 28–38, 2015.

[34] S. Venkataraman *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 374–389.

[35] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Shelter Island, NY, USA: Manning Publications Co., 2015.

[36] R. Barber *et al.*, "Wildfire: Concurrent blazing data ingest and analytics," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2077–2080.

[37] A. Costea *et al.*, "VectorH: Taking SQL-on-hadoop to the next level," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1105–1117.

[38] Y. Cao *et al.*, "ES^2: A cloud data storage system for supporting both OLTP and OLAP," in *Proc. IEEE Int. Conf. Data Eng.*, 2018.

[39] J. Ramnarayan, S. Menon, S. Wale, and H. Bhanawat, "SnappyData: A hybrid system for transactions, analytics, and streaming," in *Proc. 10th ACM Int. Conf. Distrib. Event-Based Syst.*, 2016, pp. 372–373.

[40] [Online]. Available: https://databricks.com/product/databricks-delta

[41] V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi , "Janus: A hybrid scalable multi-representation cloud datastore," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 4, pp. 689–702, Apr. 2018.

[42] S. Das *et al.*, "All aboard the Databus! Linkedin's scalable consistent change data capture platform," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 1–14.

[43] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.

[44] P. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos, "Real-time analytical processing with SQL server," *Proc. VLDB Endowment*, vol. 8, pp. 1740–1751, 2015.

[45] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso, "BatchDB: Efficient isolated execution of hybrid OLTP + OLAP workloads for interactive applications," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 37–50.

[46] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann, “ScyPer: Elastic OLAP throughput on transactional data,” in *Proc. 2nd Workshop Data Analytics Cloud*, 2013, pp. 11–15.

[47] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas, “Accordion: Elastic scalability for database systems supporting distributed transactions,” *Proc. VLDB Endowment*, vol. 7, pp. 1035–1046, 2014.

[48] E. P. C. Jones, D. J. Abadi, and S. Madden, “Low overhead concurrency control for partitioned main memory databases,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 603–614.

[49] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: Fast distributed transactions for partitioned database systems,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 1–12.

[50] A. Dragojević *et al.*, “No compromises: Distributed transactions with consistency, availability, and performance,” in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 54–70.

[51] A. Kalia, M. Kaminsky, and D. G. Andersen, “FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs,” in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 185–201.

[52] W. Cao *et al.*, “PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database,” *Proc. VLDB Endowment*, vol. 11, pp. 1849–1862, 2018.

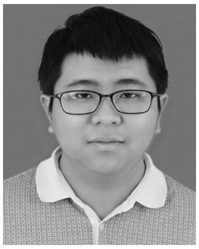
[53] [Online]. Available: https://github.com/linkedin/pinot/wiki/Architecture

[54] L. Chang *et al.*, “HAWQ: A massively parallel processing SQL engine in hadoop,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1223–1234.

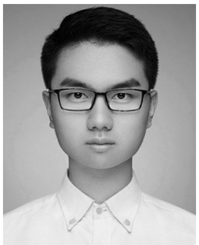
[55] J. Baker *et al.*, “Megastore: Providing scalable, highly available storage for interactive services,” in *Proc. Conf. Innovative Database Res.*, 2011, pp. 223–234.

[56] S. Das, E. Abbadi, and D. Agrawal, “ElasTraS: An elastic transactional data store in the cloud,” in *Proc. Conf. Hot Topics Cloud Comput.*, 2009, Art. no. 7.

[57] P. Pedreira, Y. Lu, S. Pershin, A. Dutta, and C. Croswhite, “Rethinking concurrency control for in-memory OLAP DBMSs,” in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1453–1464.



Zhifang Li received the bachelor’s degree from East China Normal University, Shanghai, China, in 2016, and is currently working toward the PhD degree at East China Normal University, Shanghai, China. His major interests of research include real-time analytical system, heterogenous computing system, and distributed database system.



Beicheng Peng received the bachelor’s degree from East China Normal University, Shanghai, China, in 2018, and is currently working toward the master degree at East China Normal University, Shanghai, China. His major interests of research include massively-parallel computer system, heterogenous computing, and distributed database system.

Qiuli Huang received the bachelor’s degree from the University of Shanghai for Science and Technology, Shanghai, China, in 2017. She is currently working toward the master’s degree at East China Normal University, Shanghai, China. Her research interests mainly include distributed database system.

Chuliang Weng received the PhD degree from Shanghai Jiao Tong University, Shanghai, China, in 2004. He is currently a professor with the East China Normal University (ECNU). Before joining ECNU, he worked with Huawei Central Research Institute and was an associate professor with Shanghai Jiao Tong University. He was also a visiting research scientist with Columbia University. His research interests include parallel and distributed systems, storage systems, OS, and system security.