Janus: A Hybrid Scalable Multi-Representation Cloud Datastore

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi
Department of Computer Science,
University of California, Santa Barbara
{vaibhavarora, nawab, agrawal, amr}@cs.ucsb.edu

Abstract—Cloud-based data-intensive applications have to process high volumes of transactional and analytical requests on large-scale data. Businesses base their decisions on the results of analytical requests, creating a need for real-time analytical processing. We propose Janus, a hybrid scalable cloud datastore, which enables the efficient execution of diverse workloads by storing data in different representations. Janus manages big datasets in the context of datacenters, thus supporting scaling out by partitioning the data across multiple servers. This requires Janus to efficiently support distributed transactions. In order to support the different datacenter requirements, Janus also allows diverse partitioning strategies for the different representations. Janus proposes a novel data movement pipeline to continuously ensure up to date data between the different representations. Unlike existing multi-representation storage systems and Change Data Capture (CDC) pipelines, the data movement pipeline in Janus supports partitioning and handles both distributed transactions and diverse partitioning strategies. In this paper, we focus on supporting Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) workloads, and hence use row and column-oriented representations, which are the most efficient representations for these workloads. Our evaluations over Amazon AWS illustrate that Janus can provide real-time analytical results, in addition to processing high-throughput transactional workloads.

Index Terms—Real-time analytics, Hybrid transaction and analytical processing, Data partitioning, Cloud datastores, Data movement pipeline

INTRODUCTION

The cloud computing paradigm has been widely adopted in recent years. It provides a cost effective solution for deploying new applications, removing the upfront hardware cost as well as the burden of system maintenance. The cloud computing model allows resources to be allocated on demand and provides the ability to elastically scale-out processing.

With the advent of the cloud, a plethora of web applications have also emerged. Most of these applications have to support both high frequencies of updates as well as diverse real-time analytics on data. Managing and analyzing advertising click streams, retail applications employing Just-in-time inventory management and IoT applications monitoring health data are some scenarios where insights into data are needed in real-time. In all the examples mentioned, real-time analytics must be considered while supporting high ingestion rates of inserts, updates and deletes. Applications base their decisions on analytical operations, using insights from historical data as feedback into the system. The need for a tighter feedback loop between updates and analytics has created the demand for fast real time analytical processing [16].

Owing to the different characteristics of updates and analytics, many systems use different data representations to serve them. Traditional Online transaction processing (OLTP) systems store tuples (or rows) together, referred to as row-oriented design. In contrast, column stores have been widely used for Online Analytical Processing (OLAP) workloads [14], [52]. Column-oriented design is beneficial for analytical queries due to the savings in disk bandwidth and storage cost, owing to compression. Additionally, column stores provides the ability to perform block iterations and late materialization [17]. Graph-based storage engines have been used by different systems for both transactional [9] and analytical processing [3] of networked data. Storing copies of data in different representations helps in performing the update and analytics operations on the representations most suited for them. One copy of the data can be designated as update-oriented and others as read-oriented. Some systems store data in both row and column formats to take advantage of both representations [20], [43], [47].

A major challenge for cloud-based data-intensive applications is the handling of large-scale data. Cloud datastores [22], [33] have adopted scale-out techniques as a means to support large-scale data. Data partitioning is a widely used technique for supporting scale-out, where different partitions are allocated to different servers. Hence, the data processing architectures in cloud settings need to support partitioning. Partitions are organized to limit the amount of cross-partition operations. But in cases where partitioning is not perfect, cross-partition operations have to be supported.

To employ multiple data representations in a cloud setting, data present in different representations needs to be partitioned. Due to the diverse nature of the representations and the operations being executed on them, different partitioning strategies might suit these representations. For example, an e-commerce application might want to horizontally partition the row-oriented customer data across regions, to efficiently process transactional requests. Whereas, for supporting aggregate queries, like calculating

the average revenue earned from a product across regions, vertically partitioning the column-oriented attributes might be a better choice.

A data movement pipeline is needed for moving the data between the update-oriented and read-oriented representations. Designing a data movement pipeline that provides real-time analytics in such a large-scale partitioned environment has many challenges. One of the major challenges is to maintain freshness of analytical results. Data freshness requirements mean that traditional ETL (Extract, transform and load) pipelines, used to move data hourly or daily between databases and data warehouses, are not a viable solution. Another challenge is to maintain consistency of analytical results. The ingestion of atomic updates in one representation should be atomically observed in other representations. Suppose a distributed transaction modifies an attribute in two different partitions, residing on two different servers. Any analytical query executing an aggregate operation on the attribute at the read-oriented representation should not observe the partial effect of this distributed transaction, even in the presence of differently partitioned representations. Apart from maintaining data consistency, the data movement pipeline also has to ensure that the write throughput is not affected while moving data. Since we target a cloud setting, the data movement pipeline should also be resistant to failures.

We propose and design a hybrid partitioned cloud datastore, *Janus*[1], that supports diverse workloads. It maintains copies of data in different representations. One representation supports transactional updates and the other representations are designated for analytics, and are read-only. Janus handles the execution of distributed transactions to ensure transactional consistency of operations over multiple partitions. To support different characteristics of diverse representations, Janus allows different partitioning strategies for the different representations.

Janus proposes a *data movement pipeline*, which ships data from each update-oriented partition in batches. These batches are then applied at the corresponding read-oriented partitions. Unlike existing hybrid representation storage systems [39], [43], [47] and Change Data Capture (CDC) pipelines [31], [41], the data movement pipeline in Janus supports partitioning and handles both distributed transactions and different partitioning strategies. The capturing of changes as batches at update-oriented partitions is closely integrated with the concurrency-control mechanism and the distributed commit protocol. We devise a *graph-based dependency management* algorithm for applying batches from the update-oriented partitions, across to the read-oriented partitions. The end-to-end pipeline to move the data, across the partitioned representations, is developed to ensure no disruption in transactional execution while resulting in minimal delays in the incorporation of updates into the read-oriented partitions.

In this work, we build Janus to support OLTP and OLAP workloads. Many of the proposed solutions for supporting both OLTP and OLAP workloads execute on a single server [21], [37], [43]. These solutions do not scale out, which is an essential requirement for a cloud datastore. The solutions which scale-out [26], [49] use a *single* partitioning strategy for both transactional and analytical workloads. Furthermore, most of these systems [35], [37], [40], [49] are tailored to a main-memory design, which might not be cost-effective for many cloud applications. As compared to a disk based design, a main-memory design does not support elastically growing and shrinking the memory footprint. Janus scales-out, allows different partitioning strategies for both workloads and can be integrated with both disk-based and main-memory designs.

Based on the OLTP and OLAP workloads, we use row and column-oriented representations as our update-oriented and read-oriented representations respectively. In Janus, transactional requests are executed on the row representation and analytical queries are performed on the column representation. All committed transactional updates are continuously shipped from the row partitions and consistently merged at the corresponding column partitions.

Our contributions can be summarized as follows:

- We propose a partitioned hybrid representation store, Janus, for handling diverse workloads in cloud settings. We design a data movement pipeline, which caters to partitioned data, and enables real-time analytics.
- We focus on OLTP and OLAP workloads and deploy Janus with row and column representations. The batching scheme within Janus is closely integrated with the concurrency control and distributed commit protocol at the row partitions to ensure no disruption in transactional throughput. The batch creation scheme is also integrated with the write-ahead transactional log at the row partitions, and is recoverable from failures.
- The data-movement pipeline employs a graph-based dependency management algorithm to ensure that all updates are consistently merged at the read-oriented representation, even in the presence of distributed transactions and different partitioning strategies.
- We provide an evaluation built over two open source systems, MySQL and MonetDB, and performed over Amazon AWS cloud platform. The evaluation demonstrates that Janus is capable of providing real-time analytical results while scaling-out, without affecting transactional throughput.

We give an outline of the design in Section 2. Section 3 discusses the integration of row and column partitions to support OLTP and OLAP workloads in Janus. Section 4 describes batch creation and the shipping mechanism during transaction processing in Janus. Section 5 discusses the technique to consistently merge the transactional changes to column partitions. We then present our evaluation results in Section 6 and discuss the prior work in Section 7. Section 8 discusses future work and the paper concludes in Section 9.

THE JANUS DESIGN

We now provide an overview of Janus's design, which enables the efficient execution of both transactional updates and *consistent* read-only analytical operations, at scale. Janus

1. Janus is a two-headed ancient Roman god of transitions with one head looking to the past and another to the future

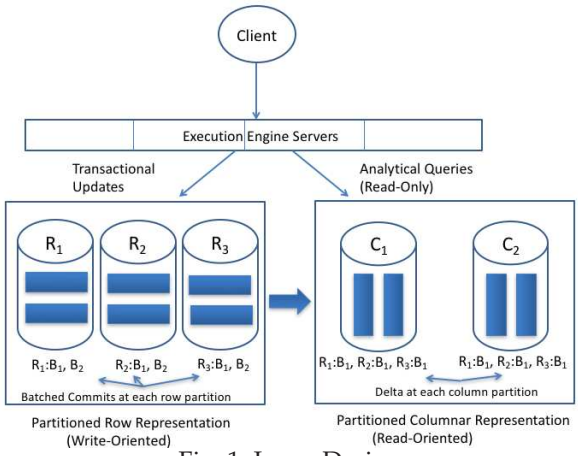


Fig. 1: Janus Design

provides serializable isolation for transactional updates. Janus also provides the guarantee that any analytical query would not observe the partial affect of any transactional update and would observe the effect of the transactions in their serialization order. Janus stores multiple copies of data in different *representations*. We divide these representations into update-oriented and read-oriented. One of the representations is used for updates and the others are read-only. Each of these representations is partitioned using a strategy appropriate for the operations supported, i.e. update or analytics. In general, these partitions can be row, column, graph etc. We briefly discuss the major components of Janus. The system design is illustrated in the Figure 1.

Execution Engine: Application clients send requests to the execution engine which then determines whether the request corresponds to an update transaction or an analytical query. Transactional update requests are routed to the update-oriented representation and the read-only analytical queries are sent to the read-oriented representation, as shown in Figure 1. The execution engine also maintains the metadata pertaining to the location of data items, which is used to route queries to the appropriate partitions. The execution engine is a scalable middleware layer, similar to ones used in large-scale partitioned and replicated environments [22], [33]. Like in [33], the execution engine is scaled by replicating the partitioning information on multiple servers, and using these multiple execution engine servers to route operations to appropriate partitions.

Update-Oriented Representation: Janus stores a copy of the data in an update-oriented representation. The update-oriented representation supports single-partition as well as cross-partition transactions. In Figure 1, Janus uses row-wise storage as update-oriented and splits the data into three partitions, R_1, R_2 and R_3. When a transaction commits, the changes made by the transaction are stored in an in-memory append only structure, referred to as a *batch*. A batch stores changes made by committed transactions at an update-oriented partition and is used to commit these changes at the read-oriented representation. The methodology for creating consistent batches is described in the Section 4.1 and 4.3.

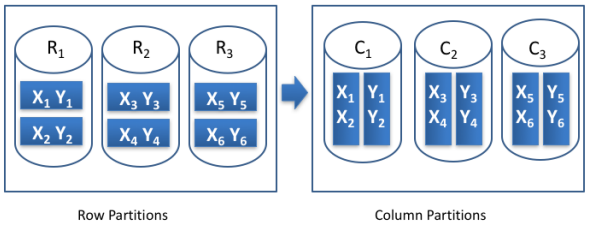
Read-Oriented Representations: Janus also stores data in other representations, which are designated as read-only. Every partition of a read-oriented representation consists of a persistent image and a *delta of changes*. The delta constitutes of incoming batches from the update-oriented representation which are yet to be ingested into the partition. In Figure 1, Janus uses column storage as read-oriented and splits a copy of the data into two partitions, C_1 and C_2. In the Figure, the row partitions and the column partitions use different partitioning strategies, and the number of row partitions (3) is different from the number of column partitions (2). Three batches arriving at C_1 are represented by R_1:B_1, R_2:B_1 and R_3:B_1. A read-oriented partition receives batches from all the update-oriented partitions that have changes mapping to that particular read-oriented partition. Updates are made to a read-oriented partition by applying the batches to the persistent image of that partition. A graph dependency management algorithm is used for applying the batches, ensuring that the consistency of data is preserved (Section 5.1).

Hybrid Partitioned Row and Column Data Management

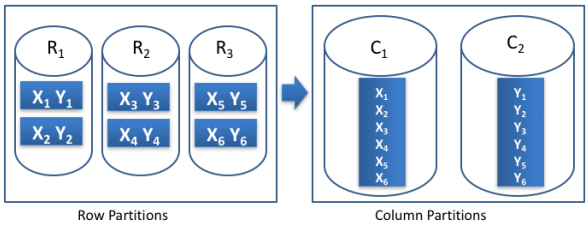
In this work, we focus on designing an instance of Janus to support OLTP and OLAP workloads. Since, row and column representations have been widely used for OLTP and OLAP workloads [17], [47], we choose these representations as update-oriented and read-oriented representations respectively. First, we discuss the partitioning of data at both the representations. Then, we explain transaction processing (Section 4) and the ingestion of the changes at the read-oriented partitions (Section 5). Although the techniques are presented for row and column, they can be applied to different representations, chosen based on the workloads being handled.

Partitioning: Janus allows applications to partition their data. Both the row and the corresponding column data can be partitioned. The row-oriented data is divided into n partitions, R_1, $R_2 \ldots R_n$ and the column data is partitioned into m partitions, C_1, $C_2 \ldots C_m$. The partitioning strategy of the columns may or may not correspond to that of the rows. Janus provides this flexibility because different applications have varied characteristics governing the analytical workloads. For example, a multi-tenant data platform will partition the data across the tenant boundaries. Hence, storing the column version of each partition is suitable, as there would not be a need for analytical operations on columns across different partitions. Whereas an e-commerce store might store customers in various regions in different row partitions. In such cases, there might be a requirement to perform analytics on columns across the row partitions to collect aggregate statistics across regions. Therefore, a partitioning strategy which stores some of the columns entirely in a single column partition (vertical partitioning) may be more efficient. Two different instances of partition mapping schemes that can be employed are:

- The column partitions can be partitioned corresponding to the row partitions. There would be n column partitions, $C_1 \ldots C_i \ldots C_n$, where C_i is the columnar version of the row partition, R_i, as represented in Figure 2(a).
- Store the entire column together in a separate partition. This would lead to r column partitions, where r is the number of attributes in the partitioned table. Each column partition, C_i would have a column



(a) Partitioning - Case 1



(b) Partitioning - Case 2

Fig. 2: Diverse Partitioning Strategies

stored in its entirety, containing data from across row partitions, as shown in Figure 2(b).

Transactional Processing

Janus handles both OLTP and OLAP workloads. OLTP operations are executed on the row partitions and column partitions are continuously updated by bulk committing the results of the updates as batches. The read-only analytical queries are then executed at column partitions.

When a transaction T arrives at the execution engine, it looks up the metadata to determine the partitions involved in the transaction. The transaction is then sent to the corresponding partitions. Each row partition employs a local concurrency control mechanism to manage the concurrent execution of transactions. In this specific implementation of Janus, each row partition uses strong strict two-phase locking (2PL) as the concurrency control method [24]. However, Janus can be adapted to other *commit-order preserving* serializable concurrency control mechanisms. The concurrency control protocol needs to be commit-order preserving to ensure the correctness of the batch generation scheme. In addition, distributed transactions employ two-phase commit (2PC) for atomic commitment across partitions. When a transaction commits, the changes made by the transaction at a partition are stored in a *batch*. We first describe the scheme for creating a batch for single-partition transactions. Then, we discuss the support for distributed transactions in Section 4.3 and describe the mechanism for sending the batches to the column partitions in Section 4.4.

Consistent Batch Generation for Single Partition Transactions

Transactional changes are stored in *batches* at every row partition. In this subsection, we consider transactions that are restricted to a single-row partition. A batch is an in-memory append only structure and comprises a set of transactional changes executed at a row partition. The batch structure is analogous to the partial transactional redo command log used for recovery by main memory databases. A tag is associated with each batch, indicating whether the batch is in an *active* or *closed* state. Each row partition has only one active batch at any instant. Once an active batch is closed, it is sent to the column-oriented representation to ingest the changes present in the batch. Each batch is also associated with a *version number* which determines the order of the batch, as compared to other batches from the same partition. The batches are continuously shipped and applied to the corresponding column partitions to keep them updated. The size of the batch shipped is a trade-off between maximizing the data freshness of analytical results and restricting the overhead involved in generating, shipping and applying the batches. To manage the size of the batch, batches are shipped after a pre-defined period. This period is named *batch shipping frequency*. The batch generation scheme ensures that all transactions executed at the row partitions are captured and the serialization order of such transactions is maintained. Each batch is also assigned a unique id, referred to as *batch-id*. The batch-id is a combination of the row partition and the batch version number. A batch with batch-id $R_i{:}B_j$ refers to changes included in batch with version number B_j from row partition R_i. In Figure 1, $R_1{:}B_1$ is a closed batch and has been shipped to the column representation. Whereas, $R_1{:}B_2$ is an active batch and any transactional changes done at R_1 will be appended to $R_1{:}B_2$.

On a commit, all the write operations of the transaction are atomically appended to the batch. All the changes at a particular row partition resulting from a single transaction are stored in the same batch. After a pre-set threshold, the batch is *marked closed* and a new batch is created and *marked active*. Transactions which commit after the old batch is closed are added to the new batch. The switch from the old batch to the new batch is done atomically. The new batch is assigned a version number, which is one more than the version number of the closed batch. Once a batch is closed it is shipped to the column partitions. Closed batches at a row partition can be discarded once the corresponding column partitions acknowledge its receipt.

Batch generation is integrated with the existing transactional log mechanism for handling failures. When a batch is closed, an additional record is written to the transactional log, noting the switch to the new batch. During the recovery phase, the batch generation scheme recovers the state of the last active batch before the crash. In addition to the existing recovery operations, the record marking the closure of the last batch is noted and is used to reconstruct the state of the active batch before the crash. Transactions with a commit record after the record marking the closed batch are added to the current active batch.

Batch Properties

The batching scheme ensures several invariant properties, needed for generating consistent batches. We now list the properties and argue that the batching scheme provides these guarantees.

Property 1. A closed batch only contains operations from committed transactions and each transaction is included in its entirety.

As each transaction is added to an active batch once it is committed, this ensures that only committed transactions are present in a batch. When a transaction is committed, all its changes are added to a batch. Thus, transactional changes

are present in their entirety for every transaction. This property ensures that all transactional updates performed at the row partitions are included in the batches.

Property 2. For any two transactions T_1 and T_2 present in batch B_i, if T_1 is serialized before T_2, $T_1 \rightarrow T_2$, then T_1 is present before T_2 in B_i.

Transactions are appended to a batch at commit. So, the order in which transactions are appended to a batch follows the commit order of transactions. This case is true if the concurrency control protocol is commit-order preserving serializable. As we are using strong strict 2PL for concurrency control, this condition is true in Janus. Hence, if $T_1 \rightarrow T_2$, then T_1 would have been appended to B_i before T_2, and as a batch is append-only, T_1 will be present before T_2 in B_i. This invariant establishes that the serialization order is preserved while appending transactions to a batch.

Property 3. For any two transactions T_1 and T_2, where T_1 is serialized before T_2, $T_1 \rightarrow T_2$, then if $T_2 \in B_i$, where B_i is a closed or an active batch, then either $T_1 \in B_i$ too or $T_1 \in B_j$, where B_j is closed and $V(B_j) < V(B_i)$, where $V(B_i)$ is the version number of batch B_i.

The order in which transactions are appended to batches follows the commit order of transactions. As T_1 is serialized before T_2, it either would have been applied to the same batch or an earlier batch. As batch version numbers increase over time, if T_1 was applied to an earlier batch, the corresponding batch would have a version number less than version number of batch B_i. This condition guarantees that batch version order corresponds to serialization order of transactions.

Property 4. If a transaction T_i belongs to a batch B_j before a failure, then after recovery from failure, T_i will still be present in batch B_j.

When a transaction commits, it is written to an active batch. If a failure is encountered, the active batch is reconstructed from the write-ahead log during recovery, by appending the update records of every transaction with a commit record after the record indicating the closure of the old batch. Since changes corresponding to any committed transaction T_i are present in the write-ahead log, any transaction T_i which belongs to the active batch B_j before the failure, will also belong to the active B_j after recovery. All the batches which have been closed have either been already shipped or they are recovered using the same protocol, hence preserving the invariant that each transaction is in the same batch as it was before the failure. This property ensures that batches are correctly recovered after failure.

These properties guarantee that the batching scheme generates consistent batches comprising all transactional changes at a row partition, even in the presence of failures.

Distributed Transaction Support

To enable operations accessing data across partitions, Janus provides the ability to perform distributed transactions. Janus employs the two-phase commit (2PC) protocol for executing distributed transactions. The execution engine acts as a coordinator of 2PC. When the transaction commits, updates of the transaction corresponding to a row partition, are stored in the batch at that partition. Hence, transactional changes of a distributed transaction may be present in multiple batches across different partitions. Furthermore, since the partitioning scheme supported at the row and column representation might be different, changes at a single column partition might correspond to batches from multiple row partitions. Consider a distributed transaction, dt, which changes a column attribute A in tuple x at R_1 and y at R_2 in Figure 2(b). The column representation uses the partitioning strategy illustrated in the figure, leading to entire column A being stored in column partition C_1. Suppose the changes done by dt are present in $R_1{:}B_1$ and $R_2{:}B_2$. Since, the changes of dt are present across different batches, the effect of these batches should be atomically visible. Hence, the algorithm for creating the batches and applying them to the column partitions needs to be carefully designed to guarantee the consistency of analytical results.

We need a method to identify the batch dependencies at the column partitions. For capturing these dependencies, *metadata* is added to each batch, which provides information about the distributed transactions present in the particular batch. The metadata includes the batch-ids of the set of batches from different row partitions, involved in distributed transactions present in the given batch. Janus integrates the bookkeeping of the metadata with the two-phase commit protocol. The needed metadata is piggybacked during the various phases of two-phase commit of a distributed transaction.

- **Prepare Phase.** During the prepare phase of two-phase commit (2PC) of any distributed transaction, each participant row partition *piggybacks* the information about the batch version number to the response of the prepare message.
- **Commit Phase.** Subsequently, if the transaction is committed, the 2PC coordinator piggybacks the batch-ids of all the batches having changes pertaining to the distributed transaction to *each* row partition, along with commit status information.

When a distributed transaction is added to a batch, the set of batches with changes pertaining to the transaction (sent by the 2PC coordinator along with the commit status), are added to the metadata of the batch. Each row partition ensures that the current active batch is not closed between sending the batch-id to the 2PC coordinator and the addition of such a transaction to the corresponding batch. In the example introduced earlier, updates and inserts corresponding to the distributed transaction dt are present in batches, $R_1{:}B_1$ and $R_2{:}B_2$. Then, $R_2{:}B_2$ is added to the metadata of batch $R_1{:}B_1$ and vice-verse. This added metadata is used to ensure that the data in column partitions remains consistent, as we describe in Section 5.1.

Batch Shipping

After a batch is closed, it is shipped to the column partitions. Batches from each row partition are sent to *all* the corresponding column partitions. This is depicted by $R_1{:}B_1$, $R_2{:}B_1$, $R_3{:}B_1$ at column partition C_1 in Figure 1. Each row partition contacts the execution engine to retrieve the metadata pertaining to the column partitions corresponding to the row partition, based on the partitioning strategies employed. This metadata is cached at the partitions. The

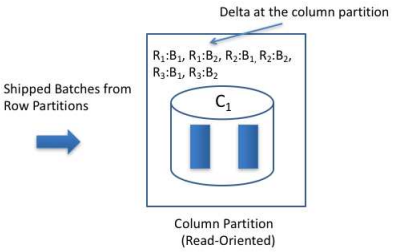


Fig. 3: A Column Partition

batches can be *pre-filtered* by dividing them into sub-batches corresponding to the different column partitions before shipping the batches. Alternatively, in an approach we name as *post-filtering*, the *entire* batch is sent to all the corresponding column partitions. Each column partition only applies the changes which correspond to that partition.

Batch Filtering. The decision of whether to post-filter or pre-filter can be based on a number of factors. The overall filtering and batch shipping cost can be divided into two parts: computation and data transfer. The computation cost is a function of cpus at the row and column partitions. In pre-filtering, the computation cost of filtering falls upon row partitions. In post-filtering, the computation cost is divided among the column partitions. The data transfer cost is a function of the available network bandwidth between row and column partition servers. As the average number of column partitions mapped from a row partition increase, the gap between the bandwidth consumed during post-filtering and that consumed in pre-filtering, increases. One the other hand, the post-filtering approach has the advantage of offloading the filtering of batches from row partitions. As Janus aims to minimize any affect on transactional throughput, Janus employs post-filtering.

Batching Threshold. A batch is closed after a fixed duration known as batch shipping frequency. The shipping frequency provides a time-based threshold to restrict the size of the batch and to ensure that batches are regularly shipped and ingested at the column partitions. Although batch shipping frequency provides a simple threshold, which can be easily adjusted, it has some drawbacks. Since, a shipping frequency only provides a time-based mechanism, it can lead to uneven batch sizes. If long running transactions with a large number of updates are present in the workload or the workload is write-heavy, this can lead to an increase in the batch size (in terms of the number of updates), which can increase the time to ingest batches at the column partitions, and thus resulting in a larger delay for an update to be reflected at the columnar representation. To avoid this bottleneck, we add the ability to combine a time-based threshold with a threshold on the number of updates in a batch (referred to as *batch shipping update-threshold*). If the number of updates in the batch goes above a certain threshold, then the batch can be closed and shipped to the column partition without waiting for the batch shipping frequency duration.

Supporting Analytical Processing

Janus executes read-only analytical queries on the column representation. In Section 4, we discussed the protocol for creating batches containing transactional changes occurring at row partitions. These batches are then sent to column representation. Each column partition consists of a persistent column-oriented copy and a delta as shown in Figure 3. The delta consists of incoming batches from different row partitions mapping to the particular column partition. We describe the protocol for merging the incoming batches to the column partition to guarantee that a read-only query accessing any *single* column partition will be consistent, even in the presence of distributed transactions at the row partitions. We then provide an extension to guarantee the consistency of analytical queries spanning *multiple* column partitions. Janus ensures that any analytical query:

- Observes the serialization order of transactions executed on the row-oriented representation
- Does not observe the partial effect of any transaction

As analytical queries can be long running, they are executed on a consistent snapshot of the data.

Consistent Single Column Partition Analytical Operations

If an incoming batch from a row partition does not have any change corresponding to a distributed transaction, then it can be applied atomically to a column partition. This ensures that analytical operations will observe the effect of each transaction present in the batch in its entirety. The update operations in an incoming batch are grouped as a single transaction. The transaction is then executed using the concurrency control scheme at the column partition. Applying the batch as a single transaction ensures that all the changes in the batch are observed atomically. Batches from each row partition are ingested in order of their version numbers. As Janus ensures that each analytical query observes a consistent snapshot of the data, any index on the column partition will be updated synchronously with the ingestion of the batch. This mechanism ensures that the serialization order of transactions executed at a single row partition is maintained at the column partition.

If distributed transactions are present in the workload and the row and column partitions are not aligned, then the changes from a distributed transaction can arrive in batches from different row partitions, as described in Section 4.3. This may lead to analytical operations not being consistent. Continuing with the example from Section 4.3, the changes done by dt are present in R_1:B_1 and R_2:B_2. R_1:B_1 includes changes to column A at tuple x, whereas R_2:B_2 includes changes to column A at tuple y. Consider the scenario where we atomically apply R_1:B_1 to the column partition and then execute an aggregate query on column A involving tuple x and y, before applying R_2:B_2. The result of such a query will be inconsistent as it would include partial changes from transaction dt. Hence, batches with partial changes from distributed transactions must be ingested atomically.

To ensure the consistency of analytical operations in the presence of distributed transactions and different partitioning strategies, changes are ingested to the column partitions by a *graph-based dependency management algorithm*. The presence of distributed transactions leads to dependencies across batches from different row partitions. Such batch dependencies are included in the metadata of each

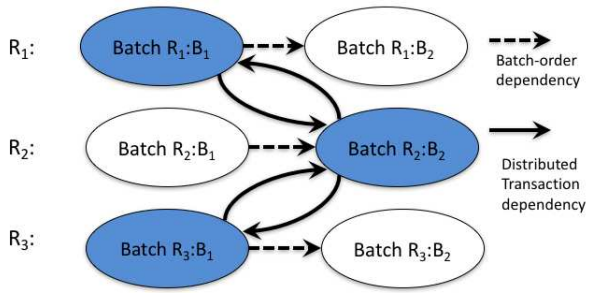


Fig. 4: Batch Dependency Graph

batch (Section 4.3). At each column partition, the batch dependencies are modeled as a directed graph, referred to as the *batch dependency graph*. As described earlier, each batch has a unique id. The id of the batch is a combination of the row partition id and the batch version number, which guarantees its uniqueness. An incoming batch, with an id α, is represented as a node with outgoing and incoming edges. An incoming edge refers to a batch upon which batch α depends on. Whereas, outgoing edges refer to the other batches dependent on batch α. The dependencies between the batches can be classified in two categories. Apart from the dependencies introduced by distributed transactions, referred to as *distributed transaction dependency*, each batch from a particular row partition is also dependent on the previous batch from the same row partition. This dependency is referred to as *batch-order dependency* and is added to capture the condition that batches from any particular row partition are ingested in their batch-id order. An example of a batch dependency graph at a particular column partition is shown in Figure 4. In the example shown in the figure, batch with id R_1:B_1 is dependent on batch with id R_2:B_2 and vice-verse. This is a dependency arising out of a distributed transaction, i.e. a distributed transaction dependency. Batch R_1:B_2 is dependent on R_1:B_1 but not vice-verse, which represents a batch-order dependency. This dependency ensures that batch R_1:B_2 can not be ingested before the batch R_1:B_1.

Any batch can only be applied to the column partition when all its dependencies are satisfied. Batch dependencies are represented by the incoming edges of the node. This condition implies that a batch can only be ingested either after all the batches it depends on are ingested or it can be ingested atomically with dependent batches. Given this condition, finding a set of batches to apply at the column partitions is equivalent to finding **strongly connected components** (SCC) in the batch dependency graph. Such a SCC should not have any incoming edge connected to any node outside the SCC. This ensures that any batch does not have any dependency apart from the batches in the SCC. Batches in such a SCC are then atomically ingested into the column partition. All the nodes of the strongly connected component can then be deleted from the batch dependency graph. In the Figure 4, batches R_1:B_1, R_2:B_2, R_3:B_1 form a strongly connected component, which will be ingested atomically after the ingestion of batch R_2:B_1. We implement the detection of SCC in the batch dependency graph using the union-find data structure [30].

The batch ingestion scheme also handles crash failures of column partitions. On ingestion of a SCC, the column partition also persists the batch-ids ingested corresponding to the row partitions it maps too. As noted in Section 4.1, a row partition only deletes a batch once it receives an acknowledgment from the corresponding column partition(s). The acknowledgement is only sent after the batch has been ingested at the column partition. On recovery from a crash, a column partition informs the corresponding row partitions of the last batch-ids ingested. The row partitions will then start sending the batches ordered after the last batch-id ingested. As the batch dependency graph at the column partition comprises the active batches, it will be reconstructed when the row partitions start sending the batches which have not been ingested at the column partition. Since, we guarantee that only ingested batches are acknowledged, row partitions will have all the batches which have not been ingested. Hence, even on crash failures, no batches will be missed and the batches will be ingested in order.

Supporting Multi Column Partition Analytical Operations

The graph-based dependency management algorithm ensures that any analytical query accessing a single column partition will return consistent results. We now provide an extension to support multi column partition read-only queries. A query which accesses data at multiple column partitions, might access an inconsistent snapshot if a batch from a row partition has only been ingested at some of the column partitions. Consider that a transaction, t, updates columns A in tuple x and column B in tuple y at R_1. Suppose, the column representation uses the partitioning strategy shown in Figure 2(b), where column A is stored in column partition C_1 for both x and y and column B is stored in C_2. Updates from transaction t are added to R_1:B_1. Now, consider a query which calculates an aggregate on column A with a select condition on column B. If the batch R_1:B_1 is only ingested at one of the column partitions, C_1, then the result of the query would be inconsistent. An inconsistency could also occur even if partitions are aligned but distributed transactions are present. Consider the distributed transaction dt, from Section 4.3. The changes done by dt are present in R_1:B_1 and R_2:B_2. R_1:B_1 includes changes to column A at tuple x, whereas R_2:B_2 includes changes to column A at tuple y. Now, suppose that column partitions use the partitioning strategy in Figure 2(a). Then R_1:B_1 would be sent to C_1 and R_2:B_2 would be sent to C_2. Any query executing at both C_1 and C_2 should atomically observe the changes in these batches.

We design a multiversioning scheme and combine it with the graph-based dependency management to ensure the efficient execution of consistent multi column partition analytical operations. Each column partition updates its state by ingesting changes from the row partitions, using the graph dependency algorithm introduced in Section 5.1. Each ingestion leads to the creation of a new version. A column partition can be represented in terms of row partitions, that map to the column partition. A column partition is represented as a *version vector* of batch versions from these row partitions. Suppose column partition C_1 maps to row partitions, R_1 and R_2 and ingests batches R_1:B_3 and R_2:B_2 to create a new version. Then, the new version at C_1 can be represented as the version vector: (B_3, B_2). Each version of the data at the column partition has a vector, known as *version vector tag* associated to it. This version vector tag is used to compare the recency of the versions. The entries of

the version vector tag of the ingested version comprise the batch version numbers corresponding to the row partitions that map to the column partition. Among two versions of data at a column partition, the version with a higher version vector tag is the newer version. The version vector tag comparison is based on the corresponding batch version numbers, which comprise the entries of the version vector tag. The information about the creation of each version is sent to the execution engine.

For each multi-column partition query, the execution engine determines the latest compatible version at each column partition involved in the query. When a multi-column partition query arrives, the execution engine first determines the column partitions involved in the query. The execution engine then determines the version with the most recent version vector tag, which corresponds to the latest ingested batch versions of all the corresponding row partitions. If some of the column partitions involved in the query receive data from a common subset of row partitions, then for each such row partition, we include the latest common batch version ingested at the column partitions. For example, suppose the query involves column partition C_1 and C_2. C_1 maps to R_1 and R_2 and C_2 maps to R_2. We determine the most recent batch version of R_1 at C_1, and the most recent version corresponding to R_2, present at both C_1 and C_2. Then, using the batch dependency metadata defined in Section 4.3, the latest version vector tag is determined, with the condition that the batch version entry (in the latest version vector tag) for each mapping row partition, satisfies all the batch dependencies. Suppose (B_3, B_2) is the most recent version vector tag corresponding to the column partitions C_1 and C_2. The batch versions are then iterated to find a version vector tag equal to or older than the tag, (B_3, B_2), which satisfies the dependencies of all batches version entries involved. Now suppose, R_1:B_3 was dependent on R_2:B_3, then the dependencies would be checked with the version vector tag (B_2, B_2). This process continues until a version vector tag is found that satisfies all the dependencies. At each column partition involved in the multi-column partition query, the query is then executed on the version corresponding to the found version vector tag. By ensuring that the batch dependencies are satisfied across column partitions, Janus guarantees that any multi-column partition query accesses a consistent version of the data.

Since this scheme can result in many versions, there is a background process which garbage collects older versions. The background job is also triggered by the execution engine, since it tracks the batch dependencies and versions maintained at each column partition. The execution engine maintains the global version of the batch dependency graph. When a Strongly Connected Component (SCC) in the graph is ingested at all the corresponding column partitions, then the versions corresponding to older batches can be garbage collected at all those column partitions. The ingestion of a SCC of the batch dependency graph signifies that a newer version (with a higher version vector tag) is available at all the dependent column partitions and all the dependencies of the newer version are satisfied. This is precisely the condition for using a particular version for an analytical query. The presence of such a version at the column partition implies that query would never be scheduled on a older version. Hence, older versions at the column partitions can be garbage collected. Suppose R_1:B_2 and R_2:B_2 form a SCC in the batch dependency graph. When batches R_1:B_2 and R_2:B_2 are ingested at all corresponding column partitions, then any version at a column partition with a version vector tag smaller than (B_2, B_2), can be garbage collected.

As the batch ingestion is handled independently by each column partition, no special recovery needs to be employed for multi-column partition queries. Each column partition recovers independently from crash failures. The multiversioning scheme determining the compatible version for a query only checks the ingested versions, and is independent of the recovery and ingestion mechanisms.

EVALUATION

Janus is evaluated using a transactional YCSB benchmark. We focus on measuring data staleness at column partitions and evaluating the impact of batching scheme on transactional throughput. First, we briefly describe the benchmark and then give information about the experimental setup, baseline and metrics collected during the experiments.

Benchmark description

We design a benchmark, which is an extension of T-YCSB (Transactional YCSB) [32]. Apart from adding the ability to invoke transactions, the benchmark is adapted to a partitioning environment and can invoke a specified percentage of distributed transactions. The Yahoo Cloud Serving Benchmark (YCSB) [28] is a benchmark for evaluating different cloud datastores. YCSB sends single key-value read and write operations to the datastore. The workloads generated by the benchmark sends multiple such operations combined as transactions. Each transaction consists of a begin operation followed by multiple read and write operations, followed by a request to commit the transaction. Each operation of a transaction is invoked as a blocking operation, i.e, the client thread blocks until it receives a response. Unless otherwise mentioned, each transaction constitutes 4 reads and a write. The percentage of writes is varied in some experiments. The benchmark also provides the ability to configure the number of clients spawned in parallel.

Workloads generated by the benchmark either comprise only single-partition transactions or a specified percentage of distributed transactions. Each transaction first picks a primary partition for the transaction. If there are no distributed transactions, then all read and write operations are uniformly distributed over the primary partition. When distributed transactions are present, each operation of a transaction has a uniform probability of accessing a partition other than the primary partition. If 10% of the transactions in the workload are distributed, then each operation has a 2% chance (since each transaction has 5 operations) of accessing a partition other than the primary partition.

To measure the performance of read-only analytical queries in Janus, the benchmark issues aggregate queries which calculate average, minimum and maximum values of an attribute.

Experimental Setup

The dataset is range partitioned over row partitions. Each row partition consists of 100,000 data items. Column

partitions employ a different range partitioning scheme with each column partition comprising 200,000 data items. Hence two row partitions correspond to a column partition. This partitioning scheme is a combination of the partitioning schemes described in Figures 2(a) and 2(b) respectively. Each experiment uses this partitioning scheme. Each item in the dataset comprises of 2 attributes: a primary key and a value. Our standard deployment consists of 20 row and 10 column partitions, along with 4 execution engine servers. As each of the row partitions, column partitions and execution engine servers is placed on a separate machine, the standard deployment employs 34 machines in total.

The evaluations were performed on AWS [1] (Amazon Web Service) EC2 (Elastic Compute Cloud). We employ m3.xlarge instances which have 4 virtual CPU cores and 15 GB memory. All machines were spawned in US East Virginia region in the same availability zone. MySQL [8] is used as the row-oriented storage engine and MonetDB [7] as the column-oriented storage engine. Note that Janus implements strict two-phase locking (2PL) concurrency control protocol, two-phase commit (2PC), as well as the batch generation and ingestion scheme. As the batch generation scheme is closely integrated with the concurrency control and the distributed commit protocol, we choose to implement both 2PL as well as 2PC at the row partitions. The implementation, therefore, does not rely on the concurrency control protocol of MySQL, but only employs it as row-oriented storage engine. On the other hand, each column partition uses MonetDB for storage as well as concurrency control. Hence, Janus employs MonetDB's optimistic concurrency control (OCC) mechanism. The concurrency control mechanism at each column partition is needed by the batch ingestion scheme to ensure the atomicity of batch ingestion. Row and column partitions reside on different EC2 machines. However, Janus can also be deployed with multiple row and column partitions being placed on the same machine. Janus is implemented in Java. It uses protocol buffers [12] for serialization and protobuf-rpc library for sending messages between the servers.

A number of application clients are spawned in parallel. Clients are co-located and uniformly distributed over the execution engine servers. Each client executes 500 transactions.

To analyze Janus's performance, we compare it against a baseline setup, where both transactions and analytics are performed on the transactional engine. To enable this setup, we turn off the batching in Janus. This comparison enables us to assess if the batching scheme affects the transactional throughput. Both setups utilize strong strict 2PL as the concurrency control protocol, ensuring an equivalent comparison.

Unless otherwise stated, the batch shipping frequency in Janus is set to 250 ms. Batch shipping frequency is the time period after which a batch is closed and shipped to the column partitions. To ensure that all row partitions do not send batches to a column partition at the same instance, we add some random noise (+/- 20%) to the batch shipping frequency at every row partition. For a shipping frequency of 250 ms, each row partition would have a frequency in the interval of [200,300] ms. Later, we also evaluate the impact of batch shipping frequency and analyze the optimal

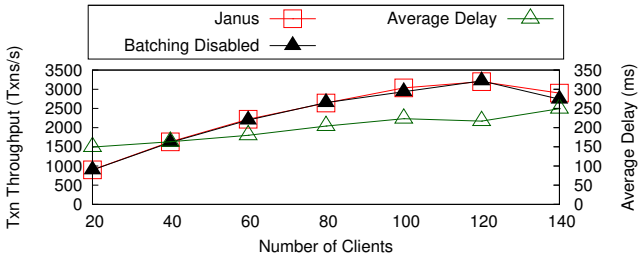


Fig. 5: Scaling Up Clients

shipping frequency for our standard deployment. We also perform an experiment where an update-based threshold is introduced for batch rollover, in addition to the batch shipping frequency. Measurements were averaged over 3 readings for smoothing any experimental variations. The measurements reported are described below.

Transactional Throughput This metric reports the number of transactions executed per second (tps).

Average Delay gives the measurement of data freshness in Janus. A delay value for a transaction includes the time period between the commitment at the row partition and its ingestion at the column partition(s). Hence, average delay includes the time period between adding the committed transaction to the batch and shipping the batch, and the time taken to ship the batch and the time for merging the batch at the column partitions. As batching is a feature of Janus and is turned off for the baseline evaluation, we only report this metric for Janus. Values reported are average of the mean delay values observed at the column partitions. The average delay metric gives a measure of freshness of the results returned by analytical queries. As row and column partitions are placed on different machines, average delay is measured across different machines. The average delay measurement faced challenges involved in measuring time in a distributed system. Initially, high clock drift values were observed at the machines. To circumvent clock drift, networking time protocol (NTP) synchronization [45] was employed. The NTP utility in unix synchronizes the clock of the server with centralized time servers.

Query Response Time is reported for read-only analytical queries performed during the experiments.

Experimental Results

Scaling Up

Figure 5 evaluates the end-to-end performance of the data movement pipeline in Janus. We employ a standard deployment of 20 row and 10 column partitions. The number of clients running concurrently were varied from 20 to 140. In this set of experiments, the workload only consists of single partition transactions. In Janus, the transactional throughput increases from around 900 tps to 3197 tps as the number of clients increase from 20 to 120. On increasing the clients to 140, the throughput decreases slightly. The baseline scenario with batching disabled also achieves similar transaction throughput illustrating that Janus's batching scheme does not adversely affect transactional performance. Less than .1% of transactions were aborted for each case. As the highest throughput is observed with 120 clients, we employ 120 clients for later experiments.

Average delay was measured for Janus. Low average delay values were observed, ranging from 149 ms with 20

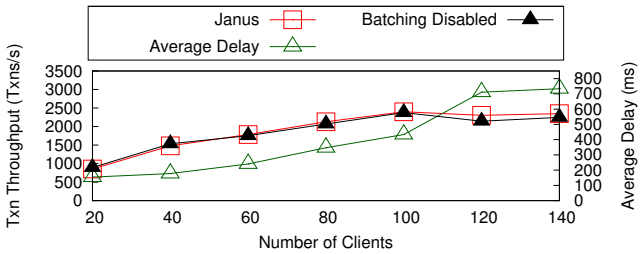


Fig. 6: Scaling Up Clients with Hot Spots

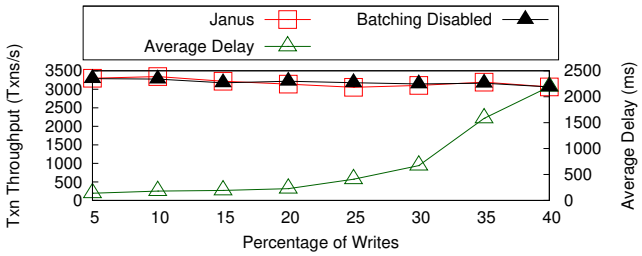


Fig. 7: Varying Read:Write Ratio

clients to 217 ms with 120 clients. The delay value increases with higher transactional throughput. As each batch comprises more transactions, it takes longer to ingest a batch into column partitions. The variance observed in delay values at different column partitions was 26.5 with 120 clients. This is low compared to the mean value of 217 ms.

These results demonstrate that Janus updates the column partitions in near real-time without impacting the transactional throughput.

Hot Spots. We also performed an experiment using a skewed workload distribution. In this experiment, 70% of the transactions access 30% of the partitions. The standard deployment of 20 row and 10 column partitions is employed and clients are scaled-up from 20 to 120. Batching still does not impact throughput and average delay was sub 750 ms, as seen in Figure 6. Delay values were higher and had greater variance due to the hot spot, as compared to results in Figure 5.

Varying Read:Write Ratio. Next, we vary the ratio of read to writes in the workload, as illustrated in Figure 7. Instead of issuing a single write for each transaction (20% writes), the overall percentage of writes in the workload is varied from 5% to 40%. The throughout slightly decreases as the percentage of writes in the workload increases. Average delay increases with the increase in percentage of writes, since each batch has more operations to ingest at column partitions. After the percentage of writes increases above 25%, delay increases at a high rate. This is because the batch shipping duration is not enough to cope with the time to ingest the batch at MonetDB, and the batches start getting queued at the column partitions. This illustrates that batch shipping frequency needs to be carefully chosen based on the workload.

Impact of Distributed Transactions

Next, Janus is evaluated with workloads comprising distributed transactions. The standard deployment of 34 machines is employed, with the number of clients set to 120. The results are presented in Figure 8. The presence of distributed transactions results in a slight decrease in throughput as Janus has to employ two-phase commit for

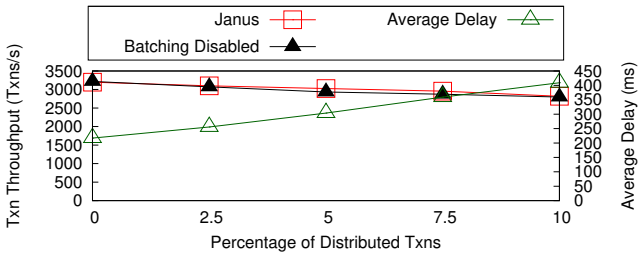


Fig. 8: Impact of Distributed Transactions

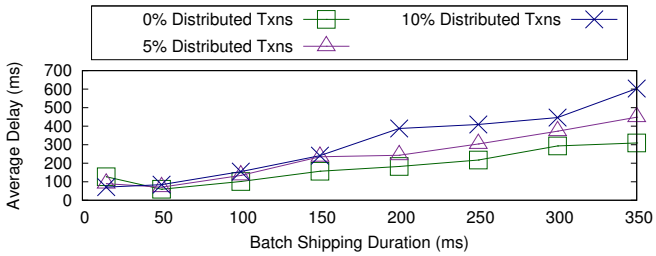


Fig. 9: Study of Batch Shipping Duration

such transactions. The results also re-illustrate that the batching scheme does not affect transactional throughput. An increase in average delay is also observed, as distributed transactions lead to dependencies among the batches. The graph-based dependency management algorithm only ingests a batch either after all the dependent batches have been ingested or with dependent batches. After all the dependencies of a batch arrive at the column partition, the batch will be a part of one of the strongly connected components (SCC) detected by the ingestion scheme, and will be merged into the column partition. The experiments illustrate that distributed transactions have impact on the data freshness. However, even in the presence of distributed transactions, Janus is capable of updating the column partitions in 304 ms and 409 ms with 5% and 10% distributed transactions respectively.

We also instrument and sub-divide the average delay duration into 4 components: average waiting time for a transaction before a batch is shipped (row-wait), network transmission and processing time (network cost), waiting time at the analytical partitions before the entire SCC has arrived (column-wait) and the actual time to ingest the batch at MonetDB (batch-ingest). For 0% distributed transactions, the row-wait duration accounts for 67% of the time and the batch-ingest duration accounts for 30% of the average delay period. The column-wait time is negligible in this case (around 1%). As the distributed transactions increase to 5%, the column-wait duration goes up-to around 5% and batch-ingest duration goes up-to 32% (batch size increases because multiple batches are ingested together to satisfy batch dependencies). With 10% distributed transactions, the dependencies among the batches cause the column-wait duration to increase to 10% of the delay duration, and the batch-ingest duration fraction goes up-to 36%. The network cost was around 1% in all the cases. The sub-division of the average-delay illustrates how the dependencies among the batches due to distributed transactions lead to the increase of average delay period.

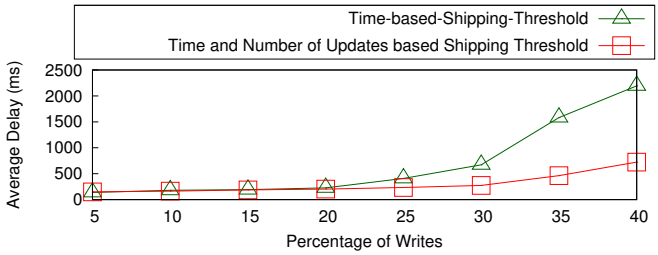


Fig. 10: Combining Batch shipping Frequency with an update-based Shipping Threshold - 0% Distributed Txns

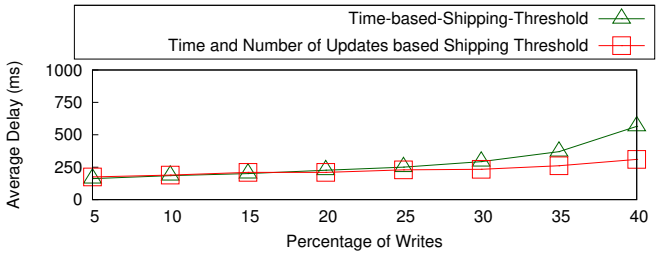


Fig. 11: Combining Batch shipping Frequency with an update-based Shipping Threshold - 5% Distributed Txns

Study of Batch Shipping Frequency

We now analyze the interplay between batch shipping frequency and average delay in Janus. The evaluation is performed using the standard deployment with 120 clients. The results are shown in Figure 9. The transactional throughput numbers were similar to the ones observed in previous experiments, and hence, are not reported.

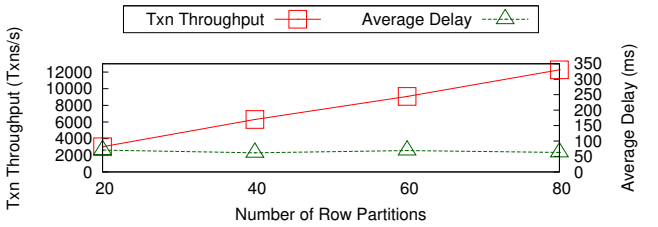
When no distributed transactions are present, as the batch shipping frequency decreases from 350 ms to 50 ms, the average delay reduces from 309 ms to 59 ms. The delay decreases because as the batch shipping period decreases, the batches are shipped more frequently and the overhead in ingesting smaller batches at the column partitions is not high enough to affect performance. But, when the batch shipping frequency is decreased further, the average delay for a transaction increases to 127 ms for a shipping frequency of 15 ms. This scenario results in MonetDB being hit with a high very update rate, which is not suited to its column-oriented design. Batch shipping frequency values below 50 ms result in higher delay values. A shipping frequency less than 50 ms does not benefit from the effect of batch committing the changes at a column partition.

The presence of distributed transactions makes Janus more sensitive to a higher batch shipping duration. Distributed transactions lead to batch dependencies and a greater shipping period might result in longer chains of dependencies, resulting in higher average delay. In the case of 10% distributed transactions, increasing the shipping period from 300 to 350 results in the average delay value increasing by 35%. When no distributed transactions are present, the same increase in the shipping period only results in a 5% increase in average delay. On the flip side, dependencies resulting from distributed transactions also result in reducing the affect of decrease in shipping duration. Waiting for batch dependencies to arrive from other row partitions, reduces the high update rate resulting from a low shipping frequency value. In the case of 5% distributed transactions, decreasing the frequency from 50 ms to 15 ms, increases the delay at a slower rate as compared to the case with no distributed transactions. This shows that as the percentage of distributed transactions in the workload increases the batch shipping frequency value should be reduced.

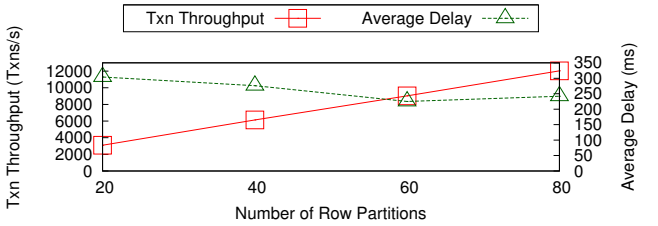
These results illustrate that the optimal value of batch shipping frequency for our standard deployment of Janus is 50 ms. Janus can update the column partitions in as little as 59 ms with no distributed transactions and 85 ms with 10% distributed transactions.

Addition of Update based Shipping Threshold

Section 4.4 discusses the bottleneck of using a time-based threshold for batch rollover, and how write-heavy workloads and long running transactions can lead to an increase in average delay. To mitigate the affect of only using a time-based threshold, Janus is integrated with the ability to specify an additional threshold to close a batch based on the number of updates (*batch shipping update-threshold*). We re-perform the experiment where the read-write ratio is varied (Figure 7), but with an additional batch shipping update-threshold, which is set to 100 updates. The threshold is set based on the number of updates present in a batch before the delay starts increasing (at 25% writes). Results in Figure 10 illustrate that adding an update-based threshold helps in reducing the increasing in delay as the percentage of writes increases.



(a) Batching Shipping Frequency: 50 ms



(b) Batch Shipping Frequency: 250 ms

Fig. 12: Scaling Out Janus

Figure 11 shows the affect of update-based threshold in the presence of distributed transactions. We observe that even with distributed transactions, the update-based threshold helps in reducing the increase in size of batch sizes as percentage of writes increases. This leads to a slower increase in delay when compared to the case where only a time-based threshold is employed. The absolute delay increases slowly with distributed transactions (as the percentage of writes increase), because the dependencies among the distributed transactions result in ingestion of multiple batches together and reducing the write-activity on MonetDB, which leads to avoiding queuing of batches at the column partitions.

Scaling Out

Figure 12 illustrates the performance of Janus while scaling-out. The percentage of distributed transactions was set at 5%. The number of clients invoked per row partition

is kept constant throughout the experiment, and increase from 120 with 20 row partitions to 480 with 80 row partitions. This ensures that the amount of contention would be the same over the entire experiment. The ratio of row to column partitions is also kept constant at 2:1, and the execution engine servers are scaled from 4 to 16. Evaluation is performed with batch shipping frequency of 250 ms as well as 50 ms (time-based threshold only). As Janus scales-out the number of partitions, throughput increases linearly and the average delay remains constant in the range of 65-70 ms with 50 ms shipping frequency, and 250-300 ms with a shipping frequency of 250 ms. These numbers illustrate that Janus's architecture is capable of scaling-out, while still updating the column partitions in real-time.

Performance of Analytical Queries

For studying analytical performance, aggregate queries calculating the average and minimum value of an attribute were run on both Janus and the baseline setup. A single column partition query calculating the average with a filter predicate took 4.6 ms on Janus and 23.2 ms on the baseline setup. A query to compute the minimum took 1.80 ms on Janus and 13.2 ms on the baseline setup. Janus performs better for analytical queries as it executes such queries on a column-oriented design, which is more suited for these queries than the row-oriented design employed by the baseline setup. As multiversioning support is not provided by MonetDB, nor by any open source column database, we did not evaluate the execution of multi-column partition queries (Section 5.2).

Related Work

Janus is motivated by and related to a wide range of research in the areas of hybrid OLTP-OLAP systems, hybrid storage layouts, data shipping and database replication.

Hybrid OLTP-OLAP Systems. Most of the proposed solutions supporting both OLTP and OLAP workloads, execute on a single server [21], [37], [43]. These solutions do not scale out, which is an essential requirement for a cloud datastore. Furthermore, most of these systems [37], [40], [51] are tailored to a main-memory design.

Hyper [40] handles both OLTP and OLAP workloads in either a column or a row-oriented *main-memory* environment, using the operating system's copy-on write technique. Virtual memory snapshots are created when read-only OLAP queries arrive. Hyper's performance is dependent on efficient forking during copy-on-write. Plattner [46] proposes supporting both transactional and analytical processing using *in-memory* column stores. Based on this vision, SAP has introduced SAP HANA [51], a main-memory engine for supporting both transactions and analytics.

Some systems have proposed adaptive layout transformation to support both OLTP and OLAP workloads in a single server setting. Arulraj et al. [21] use row, columnar and hybrid row-column representations (where a vertically partitioned group of columns are stored together) and then dynamically transform the storage layout among these representations. A multi-versioning engine is employed and each tuple is inserted in row-oriented layout. A background mechanism monitors the queries and transforms the layout by determining the attributes that are accessed together, and then grouping them. H_2o [20] also chooses from among multiple representations and a foreground layout transformation between the row, column, hybrid row-column representations. Although layout transformation can also provide the ability to take advantage of multiple representations, its performance is dependent on accurate workload prediction and can be susceptible to sudden workload shifts.

ES-2 [26] supports partitioning and stores data in a distributed file system. It uses distributed indexing and parallel sequential scans to answer OLAP queries, in addition to supporting OLTP workloads using a multiversion timestamping approach. It relies heavily on indexing and can lead to poor performance for ad hoc queries which access non-indexed data residing on the distributed file system. Unlike ES-2, Janus executes transactions and analytics on different servers as well as different representations, that are more suited for the respective workloads. ES-2 is based on a shared-storage architecture, whereas Janus employs a shared-nothing architecture.

Snappy Data [49] integrates a distributed *in-memory* data-store (GemFire) engine with Apache Spark's runtime and provides support for OLTP, OLAP and streaming. Data can be stored in *either* row or columnar form. OLTP operations are supported using the in-memory engine and OLAP operations are supported using Spark's executors. Such an architecture removes the overhead of maintaining multiple data representations. However, it cannot provide separation of transactions and analytics on different servers or different representations. Furthermore, since such an architecture uses a single data representation, it has to choose one right partitioning strategy for all workloads.

Storage in Multiple representations. Fractured Mirrors [47] was one of the first systems to store data in multiple representations. A copy of the data is stored in both the DSM (Decomposition Storage Model) layout [29], where each column is stored separately, and the row layout. A differential file is used for updating the columns. Janus is inspired from this approach and employs storage of replicas in multiple representations in a distributed setting, with differently partitioned representations. OctopusDB [34] also provides the ability to store data in multiple representations. Updates are executed by appending to a log, which acts as a primary copy of the data and multiple storage views can be incarnated in different representations. Other systems have also explored storing data in hybrid representations [4], [6], [10], [15], [39], [43], [50]. However, these approaches do not provide support for data partitioning and distributed transactions.

Column-Oriented Storage. Column stores are used for supporting OLAP workloads. MonetDB [7], Druid [2] and Vertica [14] are some available column-oriented DBMS, optimized for high volumes of query processing. C-Store [52] uses a columnar representation and is optimized for read-mostly workloads. C-Store has a write store for supporting transactional updates and a tuple mover for moving the data into the read store. The write store is also columnar and stores data in memory. The write store is limited in size, is co-located with the read store and is designed to optimize for workloads with low update rates. Some approaches aim to efficiently update column stores to support transactions [38], [48]. In contrast to Janus, these systems are primarily optimized for analytics, but are not designed

to support OLTP workloads, which have a high rate of transactional updates.

Change Data Capture [31], Pub-Sub systems [41], log shipping techniques [13], [39] and some recent ETL tools [11] have been used for continuously consuming changes from transactional systems at downstream analytical systems. This approach is amenable to scaling out and provides separation of transactions and analytics. However, such approaches provide consistency guarantees only per partition. A mechanism to overcome this is to use a single log as a source of truth. The head of the log can become a source of contention in approaches using a single distributed commit log for supporting transactions. A recent scale-out approach from SAP HANA [36] also uses a distributed commit log to support both OLAP and OLTP workloads.

Database Replication. Many replication techniques have been developed to improve the read performance of database systems, while providing some level of replica consistency guarantees. Techniques providing strong consistency [42], and 1-copy Snapshot Isolation [44] guarantees have been developed. All these systems use the same representation for the replicas. On the other hand, Janus uses different representations, that update at different rates, and also aims to attain high transactional throughput. Hence, Janus updates the read-oriented representation in batches. Due to this, the read-oriented replica can lag from the primary update-oriented copy, but provides a transactionally consistent snapshot of the data. Akal et al. [19] describe a replication technique which enables updating in batches and executing 1-copy serializable read-only transactions with given freshness guarantees. Their replication technique can support different partitioning strategies for update and read-only copies. However, it uses a global log for recording updates, which can be a bottleneck for throughput. In addition, distributed transactions are not supported.

Real-time Analytics. Recently, several architectures have been proposed for supporting both event-based processing and real-time analytical queries [5], [23], [25]. These systems do not handle transactional updates and hence, do not support OLTP workloads. Lazybase [27] uses a combination of batching and pipelined updates, and provides a trade-off between read query latency and data freshness. AIM [25] uses a PAX [18] like representation, where records are grouped into buckets and each bucket is stored column-wise. It employs differential updates and shared scans to support event-based processing and analytics in a scale-out setting. Lambda Architecture [5] stores data in two layers, a batching layer (like HDFS) optimized for batch processing and a speed layer, like a stream-processing engine, which processes data streams in real time. Each ad-hoc query is sent to both representations, and the results are then integrated.

Discussion and Future Work

As a part of our future work, we plan to extend and evaluate various aspects of Janus. The batch shipping threshold in Janus is currently set statically, which makes Janus susceptible to workload shifts. To overcome this, we aim to design schemes for dynamically setting batch shipping threshold (both shipping duration and update threshold) based on the workload and the progress of batch ingestion at the column partitions. We also plan to evaluate multi-partition analytical queries by employing analytical representations backed by multi-versioned databases. Another avenue for future research is to evaluate Janus with different representations supporting transactional and analytical workloads like graph and hybrid row-column representations.

Conclusion

Janus is a hybrid, partitioned, multi-representation datastore, for handling diverse workloads. In this paper, an instance of Janus is designed to support OLTP and OLAP workloads. Janus supports transactional requests on row-oriented storage and uses an in-memory redo log inspired batching technique to capture the transactional changes. Janus then employs a graph-based dependency management algorithm to ingest the transactional changes at the column-oriented storage. The analytical queries are executed on column storage. The devised data movement pipeline for creating, shipping and ingesting batches ensures that updates get incorporated at the column partitions with minimal delay. The data movement pipeline supports distributed transactions, as well as diversely partitioned representations. Evaluation with the transactional YCSB benchmark illustrates that Janus enables real-time analytics, while scaling-out, and without effecting transactional throughput. With 80 row partitions of data and 5% distributed transactions, the data movement pipeline of Janus is able to update the column partitions within 70 ms.

Acknowledgments. This work is supported by NSF grant IIS 1018637. Faisal Nawab is partially funded by a scholarship from King Fahd University of Petroleum and Minerals. We would also like to thank Amazon for access to Amazon EC2.

References

[1] Amazon AWS. https://aws.amazon.com/.
[2] Druid. http://druid.io/.
[3] Giraph. http://giraph.apache.org/.
[4] IBM Informix Warehouse Accelerator. http://www.iiug.org/library/ids_12/IWA%20White%20Paper-2013-03-21.pdf.
[5] Lambda Architecture. http://lambda-architecture.net/.
[6] MemSQL. http://www.memsql.com/.
[7] MonetDB. http://monetdb.com/.
[8] MySQL. https://www.mysql.com/.
[9] Neo4j. http://neo4j.com/.
[10] Oracle 12c. http://www.oracle.com/us/corporate/features/database-12c/index.html.
[11] Oracle Golden Gate. http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html.
[12] Protocol Buffers. https://developers.google.com/protocol-buffers/.
[13] Scribe. https://github.com/facebookarchive/scribe/wiki.
[14] Vertica. http://vertica.com/.
[15] Vertica FlexStore. https://www.vertica.com/2009/12/25/vertica-3-5-flexstore-the-next-generation-of-column-stores.
[16] Gartner. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/, 2014.
[17] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. of ACM SIGMOD*, pages 967–980, 2008.
[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.
[19] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proc. of VLDB*, pages 565–576, 2005.

[20] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *Proc. of ACM SIGMOD*, pages 1103–1114, 2014.

[21] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proc. of ACM SIGMOD*, pages 57–63, 2016.

[22] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, et al. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, pages 223–234, 2011.

[23] R. Barber, M. Huras, G. Lohman, C. Mohan, et al. Wildfire: Concurrent blazing data ingest and analytics. In *Proc. ACM of SIGMOD*.

[24] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[25] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, et al. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proc. of ACM SIGMOD*, pages 251–264, 2015.

[26] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, et al. Es 2: A cloud data storage system for supporting both oltp and olap. In *Proc. of IEEE ICDE*, pages 291–302, 2011.

[27] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. Lazybase: trading freshness for performance in a scalable database. In *ACM EuroSys*, pages 169–182, 2012.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of SoCC*, pages 143–154, 2010.

[29] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279, 1985.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Data structures for disjoint sets. *Introduction to Algorithms*, pages 498–524, 2001.

[31] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, et al. All aboard the databus!: Linkedin's scalable consistent change data capture platform. In *Proc. of SoCC*, page 18, 2012.

[32] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. of VLDB*, 4(8):494–505, 2011.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, et al. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220, 2007.

[34] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[35] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[36] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, et al. Towards scalable real-time analytics: an architecture for scale-out of olxp workloads. *Proc. of VLDB*, 8(12):1716–1727, 2015.

[37] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *Proc. of VLDB*, 4(2):105–116, 2010.

[38] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proc. of ACM SIGMOD*, pages 543–554, 2010.

[39] H. Jafarpour, J. Tatemura, and H. Hacigümüs. Transactional replication in hybrid data store architectures. In *Proc. of EDBT*, pages 569–580, 2015.

[40] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proc. of IEEE ICDE*, pages 195–206, 2011.

[41] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proc. of NetDB*, pages 1–7, 2011.

[42] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *Proc. of IEEE ICDE*, pages 52–63, 2010.

[43] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, et al. Real-time analytical processing with sql server. *Proc. of VLDB*, 8(12):1740–1751, 2015.

[44] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proc. of ACM SIGMOD*, pages 419–430, 2005.

[45] D. L. Mills. Network time protocol (ntp). *Network*, 1985.

[46] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proc. of ACM SIGMOD*, pages 1–2, 2009.

[47] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

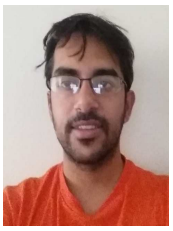
[48] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, et al. Db2 with blu acceleration: So much more than just a column store. *Proc. of VLDB*, 6(11):1080–1091, 2013.

[49] J. Ramnarayan, S. Menon, S. Wale, and H. Bhanawat. Snappydata: a hybrid system for transactions, analytics, and streaming: demo. In *Proc. of ACM DEBS*, pages 372–373, 2016.

[50] J. Schaffner, A. Bog, J. Krüger, and A. Zeier. A hybrid row-column oltp database architecture for operational reporting. In *Business Intelligence for the Real-Time Enterprise*, pages 61–74. Springer, 2009.

[51] V. Sikka, F. Färber, W. Lehner, S. K. Cha, et al. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proc. of ACM SIGMOD*, pages 731–742, 2012.

[52] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, et al. C-store: a column-oriented dbms. In *Proc. of VLDB*, 2005.



Vaibhav Arora is a doctoral student in Computer Science at the University of California, Santa Barbara. His current research interests are in the areas of data management in cloud computing environments, scalable real-time data analytics and heterogeneous data processing. He received his M.S. in Computer Science from University of California, Santa Barbara and B.Tech. in Computer Science from National Institute of Technology, Tiruchirappalli (Nit Trichy).

Faisal Nawab is a doctoral student at the University of California, Santa Barbara. His current research work is in the areas of global-scale data management, big data analytics, and data management on emerging non-volatile memory technology. He received his M.S. from King Abdullah University of Science and Technology (KAUST) and his B.S. from King Fahd University (KFUPM).

Divyakant Agrawal is a Professor of Computer Science at the University of California, Santa Barbara. His research expertise is in the areas of database systems, distributed computing, and large-scale information systems. Divyakant Agrawal is an ACM Distinguished Scientist (2010), an ACM Fellow (2012), an IEEE Fellow (2012), and Fellow of AAAS(2016). His current interests are in the areas of scalable data management and data analysis in cloud computing environments, security and privacy of data in the cloud, and scalable analytics over social networks data.

Amr El Abbadi is a Professor of Computer Science at the University of California, Santa Barbara. He received his B. Eng. from Alexandria University, Egypt, and his Ph.D. from Cornell University. Prof. El Abbadi is an ACM Fellow, AAAS Fellow, and IEEE Fellow. He currently serves on the executive committee of the IEEE Technical Committee on Data Engineering (TCDE) and was a board member of the VLDB Endowment from 2002 to 2008. He has published over 300 articles in databases and distributed systems and has supervised over 30 PhD students.