SnappyData: A Unified Cluster for Streaming, Transactions, and Interactive Analytics

Barzan Mozafari[1,2] Jags Ramnarayan[2] Sudhir Menon[2]
Yogesh Mahajan[2] Soubhik Chakraborty[2] Hemant Bhanawat[2] Kishor Bachhav[2]
[1]University of Michigan, Ann Arbor, MI [2]SnappyData Inc., Portland, OR
[1]mozafari@umich.edu
[2]{barzan,jramnarayan,smenon,ymahajan,schakraborty,hbhanawat,kbachhav}@snappydata.io

ABSTRACT

Many modern applications are a mixture of streaming, transactional and analytical workloads. However, traditional data platforms are each designed for supporting a specific type of workload. The lack of a single platform to support all these workloads has forced users to combine disparate products in custom ways. The common practice of stitching heterogeneous environments has caused enormous production woes by increasing complexity and the total cost of ownership.

To support this class of applications, we present SnappyData as the first unified engine capable of delivering analytics, transactions, and stream processing in a single integrated cluster. We build this hybrid engine by carefully marrying a big data computational engine (Apache Spark) with a scale-out transactional store (Apache GemFire). We study and address the challenges involved in building such a hybrid distributed system with two conflicting components designed on drastically different philosophies: one being a lineage-based computational model designed for high-throughput analytics, the other a consensus- and replication-based model designed for low-latency operations.

1. INTRODUCTION

An increasing number of enterprise applications, particularly those in financial trading and IoT (Internet of Things), produce mixed workloads with all of the following: (1) continuous stream processing, (2) online transaction processing (OLTP), and (3) online analytical processing (OLAP). These applications need to simultaneously consume high-velocity streams to trigger real-time alerts, ingest them into a write-optimized transactional store, and perform analytics to derive deep insight quickly. Despite a flurry of data management solutions designed for one or two of these tasks, there is no single solution that is apt for all three.

SQL-on-Hadoop solutions (e.g., Hive, Impala/Kudu and SparkSQL) use OLAP-style optimizations and columnar formats to run OLAP queries over massive volumes of static data. While apt for batch-processing, these systems are not designed as real-time operational databases, as they lack the ability to mutate data with transactional consistency, to use indexing for efficient point accesses, or to handle high-concurrency and bursty workloads. For example, Wildfire [17] is capable of analytics and stream ingestion but lacks ACID transactions.

Hybrid transaction/analytical processing (HTAP) systems, such as MemSQL, support both OLTP and OLAP queries by storing data in dual formats (row and columns), but need to be used alongside an external streaming engine (e.g., Storm [34], Kafka, Confluent) to support stream processing.

Finally, there are numerous academic [20, 31, 33] and commercial [2, 8, 14, 34] solutions for stream and event processing. Although some stream processors provide some form of state management or transactions (e.g., Samza [2], Liquid [23], S-Store [27]), they only allow simple queries on streams. However, more complex analytics, such as joining a stream with a large history table, need the same optimizations used in an OLAP engine [18, 26, 33]. For example, streams in IoT are continuously ingested and correlated with large historical data. Trill [19] supports diverse analytics on streams and columnar data, but lacks transactions. DataFlow [15] focuses on logical abstractions rather than a unified query engine.

Consequently, the demand for mixed workloads has resulted in several composite data architectures, exemplified in the "lambda" architecture, which requires multiple solutions to be stitched together — a difficult exercise that is time consuming and expensive.

In capital markets, for example, a real-time market surveillance application has to ingest trade streams at very high rates and detect abusive trading patterns (e.g., insider trading). This requires correlating large volumes of data by joining a stream with (1) historical records, (2) other streams, and (3) financial reference data which can change throughout the trading day. A triggered alert could in turn result in additional analytical queries, which will need to run on both ingested and historical data. In this scenario, trades arrive on a message bus (e.g., Tibco, IBM MQ, Kafka) and are processed by a stream processor (e.g., Storm) or a homegrown application, while the state is written to a key-value store (e.g., Cassandra) or an in-memory data grid (e.g., GemFire). This data is also stored in HDFS and analyzed periodically using a SQL-on-Hadoop or a traditional OLAP engine.

These heterogeneous workflows, although far too common in practice, have several drawbacks (D1–D4):

D1. Increased complexity and total cost of ownership: The use of incompatible and autonomous systems significantly increases their total cost of ownership. Developers have to master disparate APIs, data models, and tuning options for multiple products. Once

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

in production, operational management is also a nightmare. To diagnose the root cause of a problem, highly-paid experts spend hours to correlate error logs across different products.

D2. Lower performance: Performing analytics necessitates data movement between multiple non-colocated clusters, resulting in several network hops and multiple copies of data. Data may also need to be transformed when faced with incompatible data models (e.g., turning Cassandra's ColumnFamilies into Storm's domain objects).

D3. Wasted resources: Duplication of data across different products wastes network bandwidth (due to increased data shuffling), CPU cycles, and memory.

D4. Consistency challenges: The lack of a single data governance model makes it harder to reason about consistency semantics. For instance, a lineage-based recovery in Spark Streaming may replay data from the last checkpoint and ingest it into an external transactional store. With no common knowledge of lineage and the lack of distributed transactions across these two systems, ensuring exactly-once semantics is often left as an exercise for the application [4].

Our goal — We aim to offer streaming, transaction processing, and interactive analytics in a single cluster, with better performance, fewer resources, and far less complexity than today's solutions.

Challenges — Realizing this goal involves overcoming significant challenges. The first challenge is the drastically different data structures and query processing paradigms that are optimal for each type of workload. For example, column stores are optimal for analytics, transactions need write-optimized row-stores, and infinite streams are best handled by sketches and windowed data structures. Likewise, while analytics thrive with batch-processing, transactions rely on point lookups/updates, and streaming engines use delta/incremental query processing. Marrying these conflicting mechanisms in a single system is challenging, as is abstracting away this heterogeneity from programmers.

Another challenge is the difference in expectations of high availability (HA) across different workloads. Scheduling and resource provisioning are also harder in a mixed workload of streaming jobs, long-running analytics, and short-lived transactions. Finally, achieving interactive analytics becomes non-trivial when deriving insight requires joining a stream against massive historical data [7].

Our approach — Our approach is a seamless integration of Apache Spark, as a computational engine, with Apache GemFire, as an in-memory transactional store. By exploiting the complementary functionalities of these two open-source frameworks, and carefully accounting for their drastically different design philosophies, SnappyData is the first unified, scale-out database cluster capable of supporting all three types of workloads. SnappyData also relies on a novel probabilistic scheme to ensure interactive analytics in the face of high-velocity streams and massive volumes of stored data.

Contributions — We make the following contributions.

1. We discuss the challenges of marrying two breeds of distributed systems with drastically different design philosophies: a lineage-based system designed for high-throughput analytics (Spark) and a consensus-driven replication-based system designed for low-latency operations (GemFire) §2.

2. We introduce the first unified engine to support streaming, transactions, and analytics in a single cluster. We overcome the challenges above by offering a unified API §4, utilizing a hybrid storage engine, sharing state across applications to minimize serialization §5, providing high-availability through low-latency failure detection and decoupling applications from data servers §6.1, bypassing the scheduler to interleave fine-grained and long-running jobs §6.2, and ensuring transactional consistency §6.3.

3. Using a mixed benchmark, we show that SnappyData delivers 1.5–2× higher throughput and 7–142× speedup compared to today's state-of-the-art solutions §7.

2. OVERVIEW

2.1 Approach Overview

To support mixed workloads, SnappyData carefully fuses Apache Spark, as a computational engine, with Apache GemFire, as a transactional store.

Through a common set of abstractions, Spark allows programmers to tackle a confluence of different paradigms (e.g., streaming, machine learning, SQL analytics). Spark's core abstraction, a Resilient Distributed Dataset (RDD), provides fault tolerance by efficiently storing the lineage of all transformations instead of the data. The data itself is partitioned across nodes and if any partition is lost, it can be reconstructed using its lineage. The benefit of this approach is two-fold: avoiding replication over the network, and higher throughput by operating on data as a batch. While this approach provides efficiency and fault tolerance, it also requires that an RDD be immutable. In other words, Spark is simply designed as a computational framework, and therefore (i) does not have its own storage engine and (ii) does not support mutability semantics.[1]

On the other hand, Apache GemFire [1] (a.k.a. Geode) is one of the most widely adopted in-memory data grids in the industry,[2] which manages records in a partitioned row-oriented store with synchronous replication. It ensures consistency by integrating a dynamic group membership service and a distributed transaction service. GemFire allows for indexing and both fine-grained and batched data updates. Updates can be reliably enqueued and asynchronously written back out to an external database. In-memory data can also be persisted to disk using append-only logging with offline compaction for fast disk writes [1].

Best of two worlds — To combine the best of both worlds, SnappyData seamlessly integrates Spark and GemFire runtimes, adopting Spark as the programming model with extensions to support mutability and HA (high availability) through GemFire's replication and fine grained updates. This marriage, however, poses several non-trivial challenges.

2.2 Challenges of Marrying Spark & GemFire

Each Spark application runs as an independent set of processes (i.e., executor JVMs) on the cluster. While immutable data can be cached and reused in these JVMs within a single application, sharing data across applications requires an external storage tier (e.g., HDFS). In contrast, our goal in SnappyData is to achieve an "always-on" operational design whereby clients can connect at will, and share data across any number of concurrent connections. The first challenge is thus to alter the life-cycle of Spark executors so that their JVMs are *long-lived* and *de-coupled from individual applications*. This is difficult because, unlike Spark which spins up executors on-demand (using Mesos or YARN) with resources

[1] Although IndexedRDD [6] offers an updatable key-value store [6], it does not support colocation for high-rate ingestions or distributed transactions. It is also unsuitable for HA, as it relies on disk-based checkpoints for fault tolerance.

[2] GemFire is used by major airlines, travel portals, insurance firms, and 9 out of 10 investment banks on Wall Street [1].

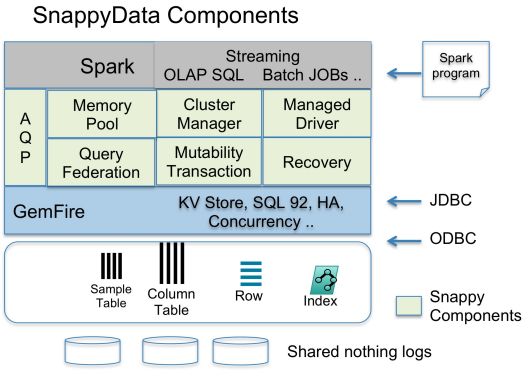


Figure 1: SnappyData's core components

sufficient only for the current job, we need to employ a static resource allocation policy whereby the same resources are reused concurrently across several applications. Moreover, unlike Spark which assumes that all jobs are CPU-intensive and batch (or micro-batch), in a hybrid workload we do not know if an operation is a long-running and CPU-intensive job or a low-latency data access.

The second challenge is that in Spark a single driver orchestrates all the work done on the executors. Given the need for high concurrency in our hybrid workloads, this driver introduces (i) a single point of contention, and (ii) a barrier for HA. If the driver fails, the executors are shutdown, and any cached state has to be re-hydrated.

Due to its batch-oriented design, Spark uses a block-based memory manager and requires no synchronization primitives over these blocks. In contrast, GemFire is designed for fine-grained, highly concurrent and mutating operations. As such, GemFire uses a variety of concurrent data structures, such as distributed hashmaps, treemap indexes, and distributed locks for pessimistic transactions. SnappyData thus needs to (i) extend Spark to allow arbitrary point lookups, updates, and inserts on these complex structures, and (ii) extend GemFire's distributed locking service to support modifications of these structures from within Spark.

Spark RDDs are immutable while GemFire tables are not. Thus, Spark applications accessing GemFire tables as RDDs may experience non-deterministic behavior. A naïve approach of creating a copy when the RDD is lazily materialized is too expensive and defeats the purpose of managing local states in Spark executors.

Finally, Spark's growing community has zero tolerance for incompatible forks. This means that, to retain Spark users, SnappyData cannot change Spark's semantics or execution model for existing APIs (i.e., all changes in SnappyData must be extensions).

3. ARCHITECTURE

Figure 1 depicts SnappyData's core components (the original components from Spark and GemFire are highlighted).

SnappyData's hybrid storage layer is primarily in-memory, and can manage data in row, column, or probabilistic stores. SnappyData's column format is derived from Spark's RDD implementation. SnappyData's row-oriented tables extend GemFire's table and thus support indexing, and fast reads/writes on indexed keys §5.1. In addition to these "exact" stores, SnappyData can also summarize data in *probabilistic* data structures, such as stratified samples and other forms of synopses. SnappyData's query engine has built-in support for approximate query processing (AQP), which can exploit these probabilistic structures. This allows applications to trade accuracy for interactive-speed analytics on streams or massive datasets §5.2.

SnappyData supports two programming models—SQL (by extending SparkSQLdialect) and Spark's API. Thus, one can perceive

```
// Create a SnappyContext from a SparkContext
val spContext = new org.apache.spark.SparkContext(conf)
val snpContext = org.apache.spark.sql. SnappyContext (
      spContext)

// Create a column table using SQL
snpContext.sql("CREATE TABLE MyTable (id int, data string)
      using column")

// Append contents of a DataFrame into the table
someDataDF.write.insertInto("MyTable");

// Access the table as a DataFrame
val myDataFrame: DataFrame = snpContext.table("MyTable")
println(s"Number of rows in MyTable = ${myDataFrame.count()
      }")
```

Listing 1: Working with DataFrames in SnappyData

SnappyData as a SQL database that uses Spark's API as its language for stored procedures. Stream processing in SnappyData is primarily through Spark Streaming, but it is modified to run *in-situ* with SnappyData's store §4.

SQL queries are federated between Spark's Catalyst and GemFire's OLTP engine. An initial query plan determines if the query is a low latency operation (e.g., a key-based lookup) or a high latency one (scans/aggregations). SnappyData avoids scheduling overheads for OLTP operations by immediately routing them to appropriate data partitions §6.2.

To support replica consistency, fast point updates, and instantaneous detection of failure conditions in the cluster, SnappyData relies on GemFire's P2P (peer-to-peer) cluster membership service [1]. Transactions follow a 2-phase commit protocol using GemFire's Paxos implementation to ensure consensus and view consistency across the cluster.

4. A UNIFIED API

Spark offers a rich procedural API for querying and transforming disparate data formats (e.g., JSON, Java Objects, CSV). Likewise, to retain a consistent programming style, SnappyData offers its mutability functionalities as extensions of SparkSQL's dialect and its DataFrame API. These extensions are backward compatible, i.e., applications that do not use them observe Spark's original semantics.

A DataFrame in Spark is a distributed collection of data organized into named columns. A DataFrame can be accessed from a `SQLContext`, which itself is obtained from a `SparkContext` (a `SparkContext` is a connection to Spark's cluster). Likewise, much of SnappyData's API is offered through `SnappyContext`, which is an extension of `SQLContext`. Listing 1 is an example of using `SnappyContext`.

Stream processing often involves maintaining counters or more complex multi-dimensional summaries. As a result, stream processors today are either used alongside a scale-out in-memory key-value store (e.g., Storm with Redisor Cassandra) or come with their own basic form of state management (e.g., Samza, Liquid [23]). These patterns are often implemented in the application code using simple get/put APIs. While these solutions scale well, we find that users modify their search patterns and trigger rules quite often. These modifications require expensive code changes and lead to brittle and hard-to-maintain applications.

In contrast, SQL-based stream processors offer a higher level abstraction to work with streams, but primarily depend on row-oriented stores (e.g., [5, 8, 27]) and are thus limited in supporting

complex analytics. To support continuous queries with scans, aggregations, top-K queries, and joins with historical and reference data, some of the same optimizations found in OLAP engines must be incorporated in the streaming engine [26]. Thus, SnappyData extends Spark Streaming to allow declaring and querying streams in SQL. More importantly, SnappyData provides OLAP-style optimizations to enable scalable stream analytics, including columnar formats, approximate query processing, and co-partitioning [9].

5. HYBRID STORAGE

5.1 Row and Column Tables

Tables can be partitioned or replicated and are primarily managed in memory with one or more consistent replicas. The data can be managed in Java heap memory or off-heap. Partitioned tables are always partitioned horizontally across the cluster. For large clusters, we allow data servers to belong to one or more logical groups, called "server groups". The storage format can be "row" (either partitioned or replicated tables) or "column" (only supported for partitioned tables) format. Row tables incur a higher in-memory footprint but are well suited to random updates and point lookups, especially with in-memory indexes. Column tables manage column data in contiguous blocks and are compressed using dictionary, run-length, or bit encoding [36]. Listing 2 highlights some of SnappyData's syntactic extensions to the `using` and `options` clauses of the `create table` statement.

We extend Spark's column store to support mutability. Updating row tables is trivial. When records are written to column tables, they first arrive in a *delta row buffer* that is capable of high write rates and then age into a columnar form. The delta row buffer is merely a partitioned row table that uses the same partitioning strategy as its base column table. This buffer table is backed by a conflating queue that periodically empties itself as a new batch into the column table. Here, conflation means that consecutive updates to the same record result in only the final state getting transferred to the column store. For example, inserted/updated records followed by deletes are removed from the queue. The delta row buffer itself uses copy-on-write semantics to ensure that concurrent application updates do not cause inconsistency [10]. SnappyData extends Spark's Catalyst optimizer to merge the delta row buffer during query execution.

5.2 Probabilistic Store

Achieving interactive response time is challenging when running complex analytics on streams, e.g., joining a stream with a large table [30]. Even OLAP queries on stored datasets can take tens of seconds to complete if they require a distributed shuffling of records, or if hundreds of concurrent queries run in the cluster [13]. In such cases, SnappyData's storage engine is capable of using probabilistic structures to dramatically reduce the volume of input data and provide approximate but extremely fast answers. SnappyData's probabilistic structures include uniform samples, stratified samples, and sketches [22]. The novelty in SnappyData's approach compared to previous AQP engines [40] is in the way that it creates and maintains these structures efficiently and in a distributed manner. Given these structures, SnappyData uses off-the-shelf error estimation techniques [11, 41]. Thus, we only discuss SnappyData's sample selection and maintenance strategies.

Sample selection — Unlike uniform samples, choosing which stratified samples to build is a non-trivial problem. The key question is which sets of columns to build a stratified sample on. Prior work has used skewness, popularity, and storage cost as the criteria for choosing column-sets [12, 13]. SnappyData extends these criteria as follows: for any declared or foreign-key join, the join key is included in a stratified sample in at least one of the participating relations (tables or streams). However, SnappyData never includes a table's primary key in its stratified sample(s). Furthermore, we offer our open-source tool, called WorkloadMiner, which automatically analyzes past query logs and reports a rich set of statistics [3]. These statistics guide SnappyData's users through the sample selection process. WorkloadMiner is integrated into CliffGuard. CliffGuard guarantees a robust physical design (e.g., set of samples), which remains optimal even if future queries deviate from past ones [28].

Once a set of samples is chosen, the challenge is how to update them, which is a key differentiator between SnappyData and previous AQP systems that use stratified samples [12, 21, 39].

Sample maintenance — Previous AQP engines that use offline sampling update and maintain their samples periodically using a single scan of the entire data [29]. This strategy is not suitable for SnappyData with streams and mutable tables for two reasons. First, maintaining per-stratum statistics across different nodes in the cluster is a complex process. Second, updating a sample in a streaming fashion requires maintaining a reservoir [16, 35], which means the sample must either fit in memory or be evicted to disk. Keeping samples entirely in memory is impractical for infinite streams unless we perpetually decrease the sampling rate. Likewise, disk-based reservoirs are inefficient as they require retrieving and removing individual tuples from disk as new tuples are sampled.

To solve these problems, SnappyData always includes timestamp as an additional column in every stratified sample. Uniform samples are treated as a special case with only one stratified column, i.e., timestamp. As new tuples arrive in a stream, a new batch (in row format) is created for maintaining a sample of each observed value of the stratified columns. Whenever a batch size exceeds a certain threshold (1M tuples by default), it is evicted and archived to disk (in a columnar format) and a new batch is started for that stratum.

Treating each micro-batch as an independent stratified sample has several benefits. First, this allows SnappyData to adaptively adjust the sampling rate for each micro-batch without the need for inter-node communications in the cluster. Second, once a micro-batch is completed, its tuples never need to be removed or replaced, and therefore they can be safely stored in a compressed columnar format and even archived to disk. Only the latest micro-batch needs to be in-memory and in row-format. Finally, each micro-batch can be routed to a single node, reducing the need for network shuffles.

5.3 State Sharing

SnappyData hosts GemFire's tables in the executor nodes as either partitioned or replicated tables. When partitioned, the individual buckets are presented as Spark RDD partitions and their access is therefore parallelized. This is similar to the way that any external data source is accessed in Spark, except that the common operators are optimized in SnappyData. For example, by keeping each partition in columnar format, SnappyData avoids additional copying and serialization and speeds up scan and aggregation operators. SnappyData can also colocate tables by exposing an appropriate partitioner to Spark (see Listing 2).

Native Spark applications can register any DataFrame as a temporary table. In addition to being visible to the Spark application, such a table is also registered in SnappyData's catalog—a shared service that makes tables visible across Spark and GemFire. This allows remote clients connecting through ODBC/JDBC to run SQL queries on Spark's temporary tables as well as tables in GemFire.

In streaming scenarios, the data can be sourced into any table

```
CREATE [Temporary] TABLE [IF NOT EXISTS] table_name (
      <column definition>
  )
  USING  [ROW | COLUMN]
    -- Should it be row or column oriented?
  OPTIONS (
     PARTITION_BY 'PRIMARY KEY | column(s) ',
        -- Partitioning on primary key or one or more columns
        -- Will be a replicated table by default
     COLOCATE_WITH 'parent_table',
        -- Colocate related records in the same partition ?
     REDUNDANCY  '1' ,
        -- How many memory copies?
     PERSISTENT [Optional disk store name]
        -- Should this persist to disk too?
     OFFHEAP "true | false",
        -- Store in off-heap memory?
     EVICTION_BY "MEMSIZE 200 | HEAPPERCENT"
       -- Heap eviction based on size or occupancy ratio ?
      ... )
```

Listing 2: Create Table DDL in SnappyData

from parent stream RDDs (DStream), which themselves could source events from an external queue, such as Kafka. To minimize shuffling, SnappyData tables can preserve the partitioning scheme used by their parent RDDs. For example, a Kafka queue listening on Telco CDRs (call detail records) can be partitioned on `subscriberID` so that Spark's DStream and the SnappyData table ingesting these records will be partitioned on the same key.

5.4 Locality-Aware Partition Design

A major challenge in horizontally partitioned distributed databases is to restrict the number of nodes involved in order to minimize (i) shuffling during query execution and (ii) distributed locks [25, 38]. In addition to network costs, shuffling can also cause CPU bottlenecks by incurring excessive copying (between kernel and user space) and serialization costs [32]. To reduce the need for shuffling and distributed locks, our data model promotes two fundamental ideas:

1. Co-partitioning with shared keys — A common technique in data placement is to take the application's access patterns into account. We pursue a similar strategy in SnappyData: since joins require a shared key, we co-partition related tables on the join key. SnappyData's query engine can then optimize its query execution by localizing joins and pruning unnecessary partitions.

2. Locality through replication — Star schemas are quite prevalent, wherein a few ever-growing fact tables are related to several dimension tables. Since dimension tables are relatively small and change less often, schema designers can ask SnappyData to replicate these tables. SnappyData particularly uses these replicated tables to optimize joins.

6. HYBRID CLUSTER MANAGER

Spark applications run as independent processes in the cluster, coordinated by the application's main program, called the driver program. Spark applications connect to cluster managers (YARN or Mesos) to acquire executor nodes. While Spark's approach is appropriate for long-running tasks, as an operational database, SnappyData's cluster manager must meet additional requirements, such as high concurrency, high availability, and consistency.

6.1 High Availability

To ensure high availability (HA), SnappyData needs to detect faults and be able to recover from them instantly.

Failure detection — Spark uses heartbeat communications with a central master process to determine the fate of the workers. Since Spark does not use a consensus-based mechanism for failure detection, it risks shutting down the entire cluster due to master failures. However, as an always-on operational database, SnappyData needs to detect failures faster and more reliably. For faster detection, SnappyData relies on UDP neighbor ping and TCP ack timeout during normal data communications. To establish a new, consistent view of the cluster membership, SnappyData relies on GemFire's weighted quorum-based detection algorithm [1]. Once GemFire establishes that a member has indeed failed, it ensures that a consistent view of the cluster is applied to all members, including the Spark master, driver, and data nodes.

Failure recovery — Recovery in Spark is based on logging the transformations used to build an RDD (i.e., its lineage) rather than the actual data. If a partition of an RDD is lost, Spark has sufficient information to recompute just that partition [37]. Spark can also checkpoint RDDs to stable storage to shorten the lineage, thereby shortening the recovery time. The decision of when to checkpoint, however, is left to the user. GemFire, on the other hand, relies on replication for instantaneous recovery, but at the cost of lower throughput. SnappyData merges these recovery mechanisms as follows:

1. Fine-grained updates issued by transactions avoid the use of Spark's lineage altogether, and instead use GemFire's eager replication for fast recovery.

2. Batched and streaming micro-batch operations are still recovered by RDD's lineage, but instead of HDFS, SnappyData writes their checkpoints to GemFire's in-memory storage, which itself relies on a fast P2P (peer-to-peer) replication for recovery. Also, SnappyData's intimate knowledge of the load on the storage layer, the data size, and the cost of recomputing a lost partition, allows for automating the choice of checkpoint intervals based on an application's tolerance for recovery time.

6.2 Hybrid Scheduler and Provisioning

Thousands of concurrent clients can simultaneously connect to a SnappyData cluster. To support this degree of concurrency, SnappyData categorizes incoming requests as low and high latency operations. By default, SnappyData treats a job as a low-latency operation unless it accesses a columnar table. However, applications can also explicitly label their latency sensitivity. SnappyData allows low-latency operations to bypass Spark's scheduler and directly operate on the data. High-latency operations are passed through Spark's fair scheduler. For low-latency operations, SnappyData attempts to re-use their executors to maximize their data locality (in-process). For high-latency jobs, SnappyData dynamically expands their compute resources while retaining the nodes caching their data.

6.3 Consistency Model

SnappyData relies on GemFire for its consistency model. GemFire supports "read committed" and "repeatable read" transaction isolation levels using a variant of the Paxos algorithm [24]. Transactions detect write-write conflicts and assume that writers rarely conflict. When write locks cannot be obtained, transactions abort without blocking [1].

SnappyData extends Spark's `SparkContext` and `SQLContext` to add mutability semantics. SnappyData gives each SQL connection its own `SQLContext` in Spark to allow applications to start, commit, and abort transactions.

While any RDD obtained by a Spark program observes a consistent view of the database, multiple programs can observe different views when transactions interleave. An MVCC mechanism (based on GemFire's internal row versions) can be used to deliver a single snapshot view to the entire application.

In streaming applications, upon faults, Spark recovers lost RDDs from their lineage. This means that some subset of the data will be replayed. To cope with such cases, SnappyData ensures the exactly-once semantics at the storage layer so that multiple write attempts are idempotent, hence relieving developers of having to ensure this in their own applications. SnappyData achieves this goal by placing the entire flow as a single transactional unit of work, whereby the source (e.g., a Kafka queue) is acknowledged only when the micro-batch is entirely consumed and the application state is successfully updated. This ensures automatic rollback of incomplete transactions.

7. EXPERIMENTS

SnappyData's main advantage is reducing the TCO by replacing disparate environments with an integrated solution for streaming, OLTP, and OLAP workloads. Since the long-term value of reduced operational costs and ease-of-use are hard to quantify, here we answer a related question: *How does* SnappyData*'s performance compare to that of existing solutions that stitch disparate but highly specialized systems for OLAP, OLTP, and stream processing?*

In summary, our comparisons against the state-of-the-art solutions indicate that, under mixed workloads, SnappyData (i) ingests data streams 3.3× faster than Spark+ Cassandra and 2× times faster than Spark+MemSQL, (ii) executes transactions 1.5× faster than Spark+ Cassandra and slightly faster than Spark+MemSQL, and (ii) runs analytical queries 142× and 7× faster than Spark+ Cassandra and Spark+MemSQL, respectively. Furthermore, when a small error is tolerable, SnappyData's probabilistic structures can deliver an additional order-of-magnitude speedup for analytical queries.

Workload — Since existing benchmarks consist of only one or two types of workloads, we present our results on a mixed workload inspired by real-world ad analytics.[3] Our workload is comprised of an ad network with three components running concurrently:

- **Streaming component.** The impression logs continuously arrive on a message bus. The ad servers aggregate these logs by publisher and geographical region, compute their average bid, number of impressions, and number of uniques every few seconds, and continuously write this data into a partitioned store.
- **Transactional component.** As new impression logs arrive, the respective profiles are updated transactionally.
- **Analytical component.** Three classes of analytical queries are executed on entire data (both past and current): (Q1) top-20 ads receiving the greatest number of impressions for each geographical region, (Q2) top-20 ads receiving the largest amount of bids for each geographical region, and (Q3) top-20 publishers receiving the largest amount of bids overall.

Baselines — We compare SnappyData with two popular Lambda stacks: Spark+Cassandra and Spark+MemSQL, whereby the mutation is handled by Cassandra (state-of-the-art KV-store) or MemSQL (state-of-the-art HTAP database), respectively. Here, the datasets are stored in Cassandra or MemSQL and exposed to Spark as RDDs. For analytical queries, the Spark-Cassandra connector fetches the required data from Cassandra (after pushing down the filters), and run the queries inside Spark. The Spark-MemSQL connector is far more aggressive and sends the analytical queries to run on MemSQL in their entirety. Both connectors provide an API for stream ingestion and data updates from the Spark context. Therefore, the actual mutation (i.e., transaction) is processed inside Cassandra and MemSQL.

Setup — We used 5 c4.2xlarge EC2 instances, each with 8 Cores, 15GB RAM, and a dedicated EBS bandwidth of 1000 Mbps. We used Kafka 2.10_0.8.2.2, Spark 2.0.0, Cassandra 3.9, MemSQL Ops-5.5.10 Community Edition, and SnappyData 0.6.1 (the latest GA versions available at the time of testing). We also used Spark-MemSQL Connector 2.10_1.3.3 and Spark-Cassandra connector 2.0.0_M3. One machine acted as Spark Master and OLAP coordinator and the other four machines were workers. A single Kafka producer process generated ad impressions (asynchronously) over 16 threads, while four Kafka brokers colocated on the worker nodes. The incoming data was processed in micro-batches by Spark Streaming and then ingested into the local store. We used 8 Kafka partitions. The Kafka producer used Avro Java objects to represent as impressions. Each impression, when serialized, was 64 bytes. Whenever supported by the store, we used columnar format for faster scans and aggregations.

Results — As shown in Figure 2a, SnappyData ingested data 2× faster than Spark+Cassandra and 1.5× faster than Spark+MemSQL. During data ingestion, all three systems updated their state transactionally. However, as shown in Figure 2b, the transaction latency was, on average, higher in Spark+Cassandra (about 0.07 milliseconds) than in Spark+MemSQL and SnappyData (about 0.04 milliseconds each).

After ingesting 300M records, we executed our analytical queries (Q1–Q3) in each system. As shown in Figure 2c, SnappyData significantly outperformed both solutions. SnappyData ran these queries on average 142× faster than Spark+ Cassandra and 7× faster than Spark+MemSQL. To determine how much of MemSQL's lower performance was due to the inefficiencies of its Spark connector, we also ran the same queries directly through MemSQL's own SQL shell. As shown in Figure 2c, MemSQL's direct performance was better than Spark+MemSQL, but it was still about 5× slower than SnappyData.

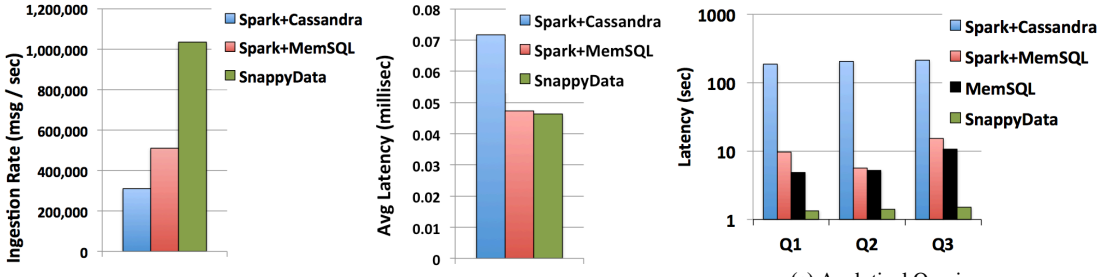
Analysis — The Spark+Cassandra connector adds a significant overhead to query processing. This is because the data has to be serialized and copied to the Spark cluster, converted into a new format, and shuffled across multiple partitions.

The Spark+MemSQL connector, on the other hand, attempts to push as much of the query processing as possible to MemSQL. This significantly reduces the amount of data movement into Spark. The connector also attempts to colocate its partitions with those of Kafka so that queuing and ingestion occur without having to shuffle any records. This explains the superior ingestion rate of Spark+MemSQL compared to Spark+Cassandra.

SnappyData embeds its column store alongside Spark executors, providing by-reference data access to rows (instead of by-copy). SnappyData also ensures that each partition at the storage layer uses its parent's partitioning method. Thus, each update becomes a local write (i.e., no shuffles). When queried, SnappyData's data is column-compressed and formatted in the same format as Spark's, leading to significantly lower latencies.

Probabilistic query performance — After ingesting 2 billion ad

[3]For comparisons using traditional benchmarks, such as TPC-H and YCSB, see [9].



(a) Streaming **(b)** Transactions **(c)** Analytical Queries

Figure 2: Performance of various solutions under a mixed workload with stream ingestion, transactions, and analytical queries.

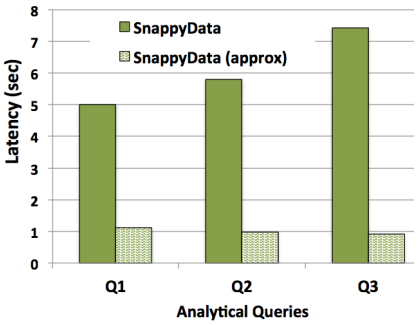


Figure 3: Interactive analytics in SnappyData w/ and w/o approximation.

impressions, we studied SnappyData's performance in its "approximate" mode (with an error tolerance of 0.05). As shown in Figure 3, our probabilistic store speeds up analytical queries by almost an additional factor of 7×.

8. CONCLUSION

We proposed a unified platform for real time operational analytics, SnappyData, to support OLTP, OLAP, and stream analytics in a single integrated solution. Our approach is a deep integration of a computational engine for high throughput analytics (Spark) with a scale-out in-memory transactional store (GemFire). SnappyData extends SparkSQL and Spark Streaming APIs with mutability semantics, and offers various optimizations to enable colocated processing of streams and stored datasets. We also made the case for integrating approximate query processing into this platform for enabling real-time operational analytics over large (stored or streaming) data. Hence, we believe that our platform significantly lowers the TCO for mixed workloads compared to disparate products that are managed, deployed, and monitored separately.

9. ACKNOWLEDGEMENTS

The first author is grateful to NSF for grants IIS-1553169, CNS-1544844, and CCF-1629397.

Bibliography

[1] Apache Geode. http://geode.incubator.apache.org/.
[2] Apache Samza. http://samza.apache.org/.
[3] CliffGuard: A General Framework for Robust and Efficient Database Optimization. http://www.cliffguard.org.
[4] Exactly-once processing with trident - the fake truth. https://www.alooma.com/blog/trident-exactly-once.
[5] IBM InfoSphere BigInsights. http://tinyurl.com/ouphdss.
[6] Indexedrdd for apache spark. https://github.com/amplab/spark-indexedrdd.
[7] Makin' Bacon and the Three Main Classes of IoT Analytics. http://tinyurl.com/zlc6den.
[8] TIBCO StreamBase. http://www.streambase.com/.
[9] Snappydata: Streaming, transactions, and interactive analytics in a unified engine. http://web.eecs.umich.edu/ mozafari/php/data/uploads/snappy.pdf, 2016.
[10] D. Abadi et al. *The Design and Implementation of Modern Column-Oriented Database Systems*. 2013.
[11] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.
[12] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[13] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. *PVLDB*, 2012.
[14] T. Akidau et al. MillWheel: fault-tolerant stream processing at internet scale. *PVLDB*, 2013.
[15] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 2015.
[16] M. Al-Kateb and B. S. Lee. Stratified Reservoir Sampling over Heterogeneous Data Streams. In *SSDBM*, 2010.
[17] R. Barber, M. Huras, G. Lohman, C. Mohan, R. Mueller, F. Özcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, et al. Wildfire: Concurrent blazing data ingest and analytics. In *SIGMOD*, 2016.
[18] L. Braun et al. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD*, 2015.
[19] B. Chandramouli et al. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 2014.
[20] S. Chandrasekaran et al. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.
[21] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.
[22] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4, 2012.
[23] R. C. Fernandez et al. Liquid: Unifying nearline and offline big data integration. In *CIDR*, 2015.
[24] J. Gray and L. Lamport. Consensus on transaction commit. *TODS*, 31(1), 2006.
[25] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 2007.
[26] E. Liarou et al. Monetdb/datacell: online analytics in a streaming column-store. *PVLDB*, 2012.
[27] J. Meehan et al. S-store: streaming meets transaction processing. *PVLDB*, 2015.
[28] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.
[29] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 2015.
[30] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, 2010.
[31] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. In *SIGMOD*, 2012.
[32] K. Ousterhout et al. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.
[33] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and

C. Zaniolo. SMM: A data stream management system for knowledge discovery. In *ICDE*, 2011.

[34] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[35] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11, 1985.

[36] R. Xin and J. Rosen. Project Tungsten: Bringing Spark closer to bare metal. http://tinyurl.com/mzw7hew.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[38] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.

[39] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, 2015.

[40] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. Abs: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[41] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.