SnappyData: Streaming, Transactions, and Interactive Analytics in a Unified Engine

Jags Ramnarayan[1] Barzan Mozafari[1,2] Sumedh Wale[1] Sudhir Menon[1]
Neeraj Kumar[1] Hemant Bhanawat[1] Soubhik Chakraborty[1]
Yogesh Mahajan[1] Rishitesh Mishra[1] Kishor Bachhav[1]

[1]SnappyData Inc., Portland, OR [2]University of Michigan, Ann Arbor, MI

[1]{jramnarayan,barzan,swale,smenon,nkumar,hbhanawat,schakraborty,ymahajan,rmishra,kbachhav}@snappydata.io
[2]mozafari@umich.edu

ABSTRACT

In recent years, our customers have expressed frustration in the traditional approach of using a combination of disparate products to handle their streaming, transactional and analytical needs. The common practice of stitching heterogeneous environments in custom ways has caused enormous production woes by increasing development complexity and total cost of ownership. With SnappyData, an open source platform, we propose a unified engine for real-time operational analytics, delivering stream analytics, OLTP and OLAP in a single integrated solution. We realize this platform through a seamless integration of Apache Spark (as a big data computational engine) with GemFire (as an in-memory transactional store with scale-out SQL semantics).

After presenting a few use case scenarios, we carefully study the challenges involved in marrying these two systems with drastically different design philosophies: Spark is a computational model designed for high-throughput analytics whereas GemFire is a transactional engine designed for low latency operations.

Moreover, we find that even in-memory solutions are often incapable of delivering truly interactive analytics (i.e., a couple of seconds), when faced with large data volumes or high velocity streams. SnappyData therefore combines state-of-the-art approximate query processing techniques and a variety of data synopses to ensure interactive analytics over both streaming and stored data. Through a novel concept of *high-level accuracy contracts (HAC)*, SnappyData is the first to offer end users an intuitive means for expressing their accuracy requirements without overwhelming them with statistical concepts.

1. INTRODUCTION

Many of our customers, particularly those active in financial trading or IoT (Internet of Things), are increasingly relying on applications whose workflows involve (1) continuous stream processing, (2) transactional and write-heavy workloads, and (3) interactive SQL analytics. These applications need to consume high-velocity streams to trigger real-time alerts, ingest them into a write-optimized store, and perform OLAP-style analytics to derive deep insight quickly.

While there have been a flurry of data management solutions designed for one or two of these tasks, there is no single solution that is apt at all three (see section 9 for a detailed survey).

SQL-on-Hadoop solutions (e.g., Hive [37], Impala [24] and Spark SQL [14]) use OLAP-style optimizations and columnar formats to run OLAP queries over massive volumes of static data. While apt at batch-processing, these systems are not designed as real-time operational databases, as they lack the ability to mutate data with transactional consistency, use indexing for efficient point accesses, or handle high-concurrency and bursty workloads.

Hybrid Transaction/Analytical Processing (HTAP) systems support both OLTP and OLAP queries by storing data in dual formats—row-oriented fashion (on disk or traditional database cache buffers) and compressed in-memory columns—but are often used alongside streaming engines (e.g., Storm, Kafka, Confluent) to support streaming processing.

Finally, stream processors (e.g., Samza [1]) provide some form of state management, but only allow for simple analytics for data streams. Complex analytics require the same optimizations used in a OLAP engine [16, 26], such as columnar formats and efficient operators for joining, grouping, or aggregating large histories. For example, according to our customers in Industrial IoT, meaningful insight often requires ingesting unbounded streams of data at very high speeds, while running continuous analytical queries on windows correlated with large quantities of history.

Consequently, the demand for mixed workloads has resulted in several composite data architectures, exemplified in the "lambda" architecture, requiring multiple solutions to be stitched together—an exercise that can be hard, time consuming and expensive.

For instance, in capital markets, a real time market surveillance application has to stream in trades 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 historical records, other streams, and financial

SIGMOD '16 June 26–July 1, 2016, San Fransisco, CA, USA
© 2016 ACM. ISBN 0-89791-88-6/97/05.
DOI: 10.1145/1235

reference data (which may change throughout the trading day). A triggered alert could in turn result in additional analytical queries, which need to run on both the ingested and historical data. Trades arrive on a message bus (e.g., Tibco, IBM MQ, Kafka) and are processed using a stream processor (e.g., Storm [38]) or a homegrown application, writing state 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 SQL-on-Hadoop OLAP engines.

Increased TCO (total cost of ownership) — This heterogeneous architecture, which is far too common among our customers, has several drawbacks (D1–D3) that significantly increase the total cost of ownership for these companies.

D1. Increased complexity: The use of incompatible and autonomous systems has significantly increased the total cost of ownership for these companies. Developers have to master disparate APIs, data models, configurations and tuning options for multiple products. Once in production, operational management is a nightmare. Diagnosing the root cause of problems often requires hard-to-find experts that have to correlate logs and metrics across different products.

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

D3. Wasted resources: With data getting duplicated, increased data shuffling wastes network bandwidth, CPU cycles and memory.

Lack of Interactive Analytics — Achieving interactive SQL analytics has remained an on-going challenge, even for modest volumes of data. Unfortunately, any analytical query that requires distributed shuffling of the records can take tens of seconds to minutes, hardly permitting interactive analytics (e.g., for exploratory analytics). Moreover, distributed clusters can be shared by hundreds of users concurrently running such queries.

Our Goal — *The challenge here is to deliver interactive-speed analytics with modest investments in cluster infrastructure and far less complexity than today.* SnappyData aims to fulfill this promise by (i) enabling streaming, transactions and interactive analytics in a single unifying system—rather than stitching different solutions—and (ii) delivering true interactive speeds via a state-of-the-art approximate query engine that can leverage a multitude of synopses as well as the full dataset.

Our Approach — We envision a single unified, scale out database cluster that ingests static data sets (e.g., from HDFS), acquires updatable reference data from enterprise databases, manages streams in memory, while permitting both continuous SQL analytics on the streams and interactive queries on entire data (acquired from streams, HDFS or enterprise DBs). To achieve this goal, our approach consists of a deep integration of Apache Spark, as a computational framework, and GemFire, as an in-memory transactional store, as described next.

Best of two worlds — Spark offers an appealing programming model to both modern application developers and data scientists. Through a common set of abstractions, Spark programmers can 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 the lineage information. The benefit of this approach is avoiding replication over the network and operating on data as a batch for higher throughput. 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.

On the other hand, GemFire is an in-memory data grid, which manages records in a partitioned row-oriented store with synchronous replication. It ensures consistency by integrating a *dynamic group membership service* (GMS) and a *distributed transaction service* (DTS). Data can be indexed and updated in a fine grained or batch manner. Updates can be reliably enqueued and asynchronously written back out to an external database. Data can also be persisted on disk using append-only logging with offline compaction for fast disk writes.

Therefore, to combine the best of both worlds, SnappyData seamlessly fuses the 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. For instance, when ingesting a stream, we process the incoming stream as a batch, avoid replication, and replay from the source on a failure. Here, the processed state could be written into the store in batches to avoid a tuple-at-a-time replication. Recovery from failure will thus be limited to the time needed to replay a single batch.

Challenges — Spark is designed as a computational engine for processing batch jobs. Each Spark application (e.g., a Map-reduce job) runs as an independent set of processes (i.e., executor JVMs) on the cluster. These JVMs are reused for the lifetime of the application. While, data can be cached and reused in these JVMs for a single application, sharing data across applications or clients requires an external storage tier, such as HDFS. We, on the other hand, target a real-time, "always-on", operational design center—clients can connect at will, and share data across any number of concurrent connections. This is similar to any operational database on the market today. Thus, to manage data in the same JVM, our first challenge is to alter the life cycle of these executors so that they are *long-lived* and *de-coupled from individual applications.*

A second but related challenge is Spark's design for how user requests (i.e., jobs) are handled. A single driver orchestrates all the work done on the executors. Given our need for high concurrency and a hybrid OLTP-OLAP workload, this driver introduces (i) a single point of contention for all requests, and (ii) a barrier for achieving high availability (HA). Executors are shutdown if the driver fails, requiring a full refresh of any cached state.

Spark's primary usage of memory is for caching RDDs and for shuffling blocks to other nodes. Data is managed in blocks and is immutable. On the other hand, we need to manage more complex data structures (along with indexes) for point access and updates. Therefore, another challenge is merging these two disparate storage systems with little impedance to the application. This challenge is exacerbated by current limitations of Spark SQL—mostly related to mutability characteristics and conformance to SQL.

Finally, Spark's strong and growing community has zero tolerance for incompatible forks. This means that no changes can be made to Spark's execution model or its semantics for existing APIs. In other words, our changes have to be an extension.

Contributions — SnappyData makes the following contributions to deliver a unified and optimized runtime.

(a) **Marrying an operational in-memory data store with Spark's computational model.** We introduce a number of extensions to fuse our runtime with that of Spark. Spark executors run in the same process space as our store's execution threads, sharing the same pool of memory. When Spark executes tasks in a partitioned manner, it is designed to keep all the available CPU cores busy. We extend this design by allowing low latency and fine grained operations to interleave and get higher priority, without involving the scheduler. Furthermore, to support high concurrency, we extend the runtime with a "Job Server" that decouples applications from data servers, operating much in the same way as a traditional database, whereby state is shared across many clients and applications. (See section 5).

(b) **Unified API for OLAP, OLTP, and streaming.** Spark builds on a common set of abstractions to provide a rich API for a diverse range of applications, such as MapReduce, Machine learning, stream processing, and SQL. While Spark deserves much of the credit for being the first of its kind to offer a unified API, we further extend its API to (i) allow for OLTP operations, e.g., transactions and inserts/updates/deletions on tables, (ii) be conformant with SQL standards, e.g., allowing tables alterations, constraints, indexes, and (iii) support declarative stream processing in SQL. (See section 4.)

(c) **Optimized Spark applications** Our goal is to eliminate the need for yet another external store (e.g., a KV store) for Spark applications. With a deeply integrated store, SnappyData improves overall performance by minimizing network traffic and serialization costs. In addition, by promoting colocated schema designs (tables and streams) where related data is colocated in the same process space, SnappyData eliminates the need for shuffling altogether in many scenarios. We describe and evaluate these optimizations in sections 7 and 8.

(d) To deliver analytics at truly interactive speeds, we have equipped SnappyData with state-of-the-art AQP techniques, as well as a number of novel features. SnappyData is the first AQP engine to (i) provide automatic bias correction for arbitrarily complex SQL queries, and (ii) provide an intuitive means for end users to express their accuracy requirements as *high-level accuracy contracts (HAC)*, without overwhelming them with numerous statistical concepts. Finally, unlike traditional load shedding techniques that are restricted to simple queries, (iii) SnappyData can provide error estimates for arbitrarily complex queries on streams.

After reviewing our use case scenarios in section 2.1, we provide a system overview in section 3. We present our data model in section 4, our cluster manager in section 5, and our AQP features in section 6. Additional optimizations offered by SnappyData are described in section 7, followed by our experimental results in section 8. Finally, we review the related work and conclude in sections 9 and 10, respectively.

2. TARGET WORKLOAD

2.1 Use Case Scenarios

Market Surveillance — Trading in financial markets has become almost entirely algorithmic and machine driven. In this environment, financial firms need to be able to detect abusive, collusive, and rogue trading and flag them in real-time before more damage is done. This requires surveillance systems to ingest post transactional data streams, analyze trades over a specific time window, and correlate with previous time windows involving the same set of instruments and groups of subscribers in order to quickly decide which trades to flag for further inspection. Data involved in market surveillance includes (i) streaming time series post transactional data, (ii) reference data, which can be updated by transactions, and (iii) historical data regarding a large number of instruments (e.g., multiple terabytes). These datasets are processed by streaming, OLTP and analytical queries. SnappyData offers an integrated solution to this problem by supporting transactional updates on reference data, while analyzing incoming streams joined with large historical datasets. Moreover, when faced with a sudden burst of incoming streams, SnappyData can still provide interactive speeds by resorting to approximate results accompanied with accuracy guarantees. For example, computing an activity's exact risk score is unnecessary as long as its approximated value is accurate enough to establish its relative position with respect to the acceptable risk level.

Location based services from telco network providers —The global proliferation of mobile devices has created a growing market for location based services. In addition to locality-aware search and navigation, network providers are increasingly relying on location-based advertising, emergency call positioning, road traffic optimization, efficient call routing, triggering preemptive maintenance of cell towers, roaming analytics, and tracking vulnerable people[35] in real time. Telemetry events are delivered as Call Detail Records (CDR), containing hundreds of attributes about each call. Ingested CDRs are cleansed and transformed for consumption by various applications. Not being able to correlate customer support calls with location specific network congestion information is a problem that frustrates customers and network technicians alike. The ability to do this in real time may involve expensive joins to history, tower traffic data and subscriber profiles. Incoming streams generate hundreds of aggregate metrics and KPIs (key performance indicators) grouped by subscriber, cell phone type, cell tower, and location. This requires continuous updates to counters accessed through primary keys (such as the subscriberID). While the generated data is massive, it still needs to be interactively queried by a data analyst for network performance analysis.

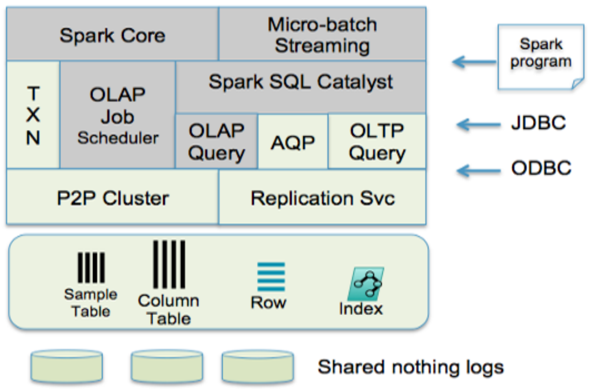

Figure 1: SnappyData's core components

Location-based services represent another common problem among our customers that involves high concurrency, continuous data updates, complex queries, time series data, and a source that cannot be throttled.

2.2 Design Assumptions

Based on the above use case scenarios, we design SnappyData based on the following assumptions.

Operational real-time data analytics — Spark is well designed for periodic, batch-centric workloads. While we retain all of Spark's functionalities, we focus more on interactive and streaming workloads. In fact, SnappyData must resemble an *"always on" operational database that is capable of concurrently serving both low-latency OLTP requests and OLAP-style analytics.*

Terabytes not Petabytes — Similar to Spark, SnappyData manages datasets primarily in main-memory. Currently, we do not target workloads with extremely large volumes. In the near term, we anticipate most workloads using SnappyData to be less than 50–100TB. Based on our experience with enterprise customers, provisioning DRAM at this scale is currently deemed cost prohibitive.

Micro-batch stream processing — We are not targeting streaming use cases that require very low latency event-at-a-time processing (e.g., high frequency algorithmic trading). Instead, we use the micro-batch approach of Spark Streaming, which is geared towards high throughput and stream processing at a second's granularity. Per-event stream processing will be particularly unrealistic since our target workloads involve complex stream analytics, which may require joins and aggregations with historical data.

3. SYSTEM OVERVIEW

This section presents a high level overview of SnappyData's core components, as well as our data pipeline as streams are ingested into our in-memory store and subsequently interacted with and analyzed.

3.1 System Architecture

Figure 1 depicts SnappyData's core components (Spark's original components are highlighted in gray).

The storage layer is primarily in-memory and manages data in either row or column formats. The column format is derived from Spark's RDD caching implementation and allows for compression. Row oriented tables can be indexed on keys or secondary columns, supporting fast reads and writes on index keys (sections 4.1).

We support two primary programming models—SQL and Spark's API. SQL access is through JDBC/ODBC and is based on Spark SQL dialect with several extensions. One could perceive SnappyData as a SQL database that uses Spark API as its language for stored procedures. We provide a glimpse over our SQL and programming APIs (section 4.2). Our stream processing is primarily through Spark Streaming, but it is integrated and runs *in-situ* with our store (section 4.3).

The OLAP scheduler and job server coordinate all OLAP and Spark jobs and are capable of working with external cluster managers, such as YARN or Mesos. We route all OLTP operations immediately to appropriate data partitions without incurring any scheduling overhead (sections 5 and 7).

To support replica consistency, fast point updates, and instantaneous detection of failure conditions in the cluster, we use a P2P (peer-to-peer) cluster membership service that ensures view consistency and virtual synchrony in the cluster. Any of the in-memory tables can be synchronously replicated using this P2P cluster (section 5).

In addition to the "exact" dataset, data can also be summarized using *probabilistic* data structures, such as stratified samples and other forms of synopses. Using our API, applications can choose to trade accuracy for performance. SnappyData's query engine has built-in support for approximate query processing (AQP) and will exploit appropriate probabilistic data structures to meet the user's requested level of accuracy or performance (section 6).

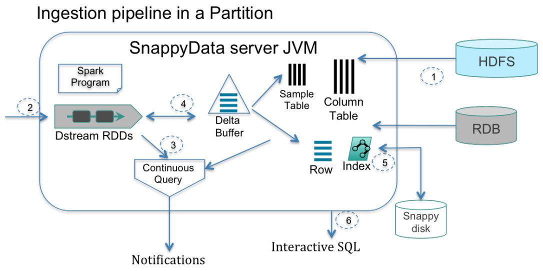

Figure 2: Data ingestion pipeline in SnappyData

3.2 Data Ingestion Pipeline

The use cases explored in section 2.1 share a common theme of stream ingestion and interactive analytics with transactional updates. The steps to support these tasks are depicted in Figure 2, and explained below.

Step 1. Once the SnappyData cluster is started and before any live streams can be processed, we ensure that the historical and reference datasets are readily accessible. The data sets may come from HDFS, enterprise relational databases (RDB), or disks managed by SnappyData. Immutable batch sources (e.g., HDFS) can be loaded in parallel into a columnar format table with or without compression. Reference data that is often mutating can be managed as row tables.

Step 2. We rely on Spark Streaming's parallel receivers to consume data from multiple sources. These receivers produce a DStream, whereby the input is batched over small time intervals and emitted as a stream of RDDs. This batched data is typically transformed, enriched and emitted as one or more additional streams. The raw incoming stream may be persisted into HDFS for batch analytics.

Step 3. Next, we use SQL to analyze these streams. As DStreams (RDDs) use the same processing and data model as data stored in tables (DataFrames), we can seamlessly combine these data structures in arbitrary SQL queries (referred to as continuous queries as they execute each time the stream emits a batch). When faced with complex analytics or high velocity streams, SnappyData can still provide answers in real time by resorting to approximation.

Step 4. The stream processing layer can interact with the storage layer in a variety of ways. The enriched stream can be efficiently stored in a column table. The results of continuous queries may result in several point updates in the store (e.g., maintaining counters). The continuous queries may join, correlate, and aggregate with other streams, history or reference data tables. When records are written into column tables one (or a small batch) at a time, data goes through stages, arriving first into a delta row buffer that is capable of high write rates, and then aging into a columnar form. Our query sub-system extends Spark's Catalyst to merge the delta row buffer during query execution.

Step 5. To prevent running out of memory, tables can be configured to evict or overflow to disk using an LRU strategy. For instance, an application may ingest all data into HDFS while preserving the last day's worth of data in memory.

Step 6. Once ingested, the data is readily available for interactive analytics using SQL. Similar to stream analytics, SnappyData can again use approximate query processing to ensure interactive analytics on massive historical data in accordance to users' requested accuracy.

4. DATA MODEL

4.1 Row and Column Oriented 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". Tables can be restricted to a subset of the nodes by specifying the groups they belong to. The storage format can be "row" (either partitioned or replicated tables) or "column" (only supported for partitioned tables) format. While a row formatted table incurs higher in-memory storage costs (with any record being a key hash lookup away) this is well suited for OLTP scenarios where random updates and deletes or point lookups are common. In-memory indexes provide further optimization for row tables. Column tables manage column data in contiguous memory and can be compressed using dictionary, run-length, or bit encodings [39]. We extend Spark's column store to support mutability.

Writing to column tables — When records are written into column tables one (or a small batch) at a time, they go through stages; first arriving into 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 delta 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 and asynchronous transfers to the column store do not cause inconsistency [7]. Our query sub-system extends Spark's Catalyst optimizer to merge the delta row buffer during query execution.

4.2 Unified API

Spark provides a rich procedural API to query, transform and work with disparate data models (e.g., JSON, Java Objects, CSV and SQL). To simplify and retain a consistent programming style, SnappyData hides the native GemFire API and instead, offers its additional functionalities as extensions to Spark SQL and the DataFrame API. The SQL extensions add support for mutability and follow the SQL standard. Some of SnappyData-specific configurations are either specified at cluster startup or via SQL DDL (Data Definition Language) extensions. These extensions are completely compatible with Spark; applications that do not use our extensions, will observe Spark's original semantics.

Below is the syntax that highlights some of the key extensions to `create table` to exploit the data model offered by SnappyData.

```
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 1: Create Table DDL in SnappyData

In Spark, a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations [14]. Any table is accessible as a DataFrame, and any DataFrame can be registered as a table. A DataFrame can be accessed from a `SQLContext`, which itself is obtained from a `SparkContext` (a `SparkContext` represents a connection to the Spark cluster). Most of SnappyData's extension API is offered through a `SnappyContext`, which is an extension of the `SQLContext`. Below is an example of working with DataFrames using the `SnappyContext`, showing how to access a table as a DataFrame, create a table using a DataFrame, and append state from a DataFrame to a row table.

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

 //Create table using SQL and access as DataFrame
snContext.sql(" CREATE TABLE MyTable ......")
myDataFrame: DataFrame = snContext.table("MyTable")

//Create a new ROW table using dataFrame 'myDataFrame'
snContext.createExternalTable(tableName, "column",
        airlineDataFrame.schema, props)
         myDataFrame.schema, props );

//Append contents of DataFrame into ROW table
someDataDF.write.format("ROW").mode(SaveMode.Append)
         .options(props).saveAsTable("T1");
```

Listing 2: Working with DataFrames

4.3 SQL-based Stream Processing

The use of a scale-out in-memory key-value stores when processing streams is pervasive, e.g., using Redis or Cassandra with Storm. A common pattern we have observed is summarizing streams either using counters on different attributes over fixed time intervals or using more complex, multi-dimensional summaries through custom programs. These patterns are often implemented in the application program with simple get/put requestsÊto the key-value store. While these solutions scale well, we also find that users modify their search patterns and trigger rules on these streams quite often. These modifications require expensive code changes, often leading to brittle, hard to maintain systems.

In contrast, SQL-based stream processors offer a richer, higher level abstraction to work with streams. Majority of these products on the market are commercial, and also primarily depend on external stores [2, 6]. Their built-in storage engines are row-oriented and typically limited in scale. As mentioned before, several of our use cases require continuous queries with joins, scans, aggregations, top-K queries, and complex correlations that involve historical and reference data. Thus, to ensure scalability of stream analytics, we believe that some of the same optimizations found in OLAP databases must be incorporated in streaming egines as well [26]. SnappyData therefore extends Spark Streaming with the following optimizations:

1. OLAP optimizations —By integrating and colocating stream processing with our hybrid in-memory storage engine, we leverage our optimizer and column store for expensive scans and aggregations, while providing fast key-based operations with our row store.

2. Reduced shuffling through co-partitioning —With SnappyData, the partitioning key used by the input queue (e.g., for Kafka sources), the stream processor and the underlying store can all be the same. This dramatically reduces the need to shuffle records.

3. Approximate stream analytics —When the volumes are too high, a stream can be summarized using various forms of samples and sketches (see section 6) to enable fast time series analytics. This is particularly useful when applications are interested in trending patterns, for instance, rendering a set of trend lines in real time on user displays [32].

4. SQL support. — To realize our goal of lowering the TCO, we extended Spark Streaming so that streams can be declared and processed using SQL. Below is an example for defining streams using SQL-like syntax. Here, we parallely ingest micro-batches from Kafka, transform the stream tuples to comply with a schema and ingest them into a column table (with possibly one or more stratified samples incrementally maintained). A "stream table" is accessible to the application as a DStream.

```
CREATE STREAM TABLE [IF NOT EXISTS] table_name (
 <column definition>
)
USING kafka_stream
OPTIONS (
 storagelevel,
 zkQuorum ,
 groupId,
 topics,
 streamToRow
)
```

Listing 3: Stream Table DDL

A SQL query that involves a "stream table" is called a continuous query (CQ) and is continuously executed as the stream emits batches. When a CQ is registered from the application code, it returns a `SchemaDStream` (an extension to `DStream` that is tied to a specific schema). We extended the Spark SQL syntax to add support for "stream table" and window semantics. Below is an example illustrating a windowed CQ within an application code:

```

val resultSet = strSnapCtx.registerCQ("
 select retweets, max(retweets) from tweetstreamTable
 window (duration '10' seconds, slide '10' seconds)
 group by retweets")

resultSet .foreachRDD(rdd => {
     val dataFrame = strSnapCtx
     .createDataFrame(rdd, resultSet.schema)

     dataFrame.write.format("column")
     .mode(SaveMode.Append)
     .saveAsTable("externalTable")
  }
)
```

Listing 4: Continuous queries on streams in SnappyData

5. HYBRID CLUSTER MANAGER

As shown in Figure 3, 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 (e.g., YARN and Mesos) to acquire executors on nodes in the cluster. Executors are processes that run computations and store data for the running application. The driver program owns a singleton (`SparkContext`) object which it uses to communicate with its set of executors.

While Spark's approach is appropriate for compute-heavy tasks scanning large datasets, SnappyData must meet additional requirements (R1–R4) as an operational database.

R1. High concurrency — SnappyData use cases involve a mixture of compute-intensive workloads and low latency (sub-millisecond) OLTP operations such as point lookups (index-based search), and insert/update of a single record. The fair scheduler of Spark is not designed to meet the low latency requirements of such operations.

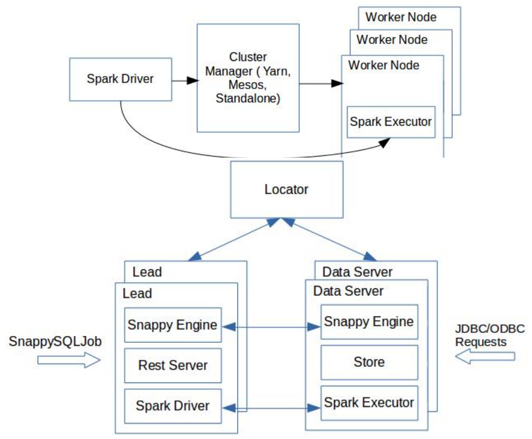

Figure 4: SnappyData's cluster architecture

R2. State sharing — Each application submitted to Spark works in isolation. State sharing across applications requires an external store, which increases latency and is not viable for near real time data sharing.

R3. High availability (HA) — As a highly concurrent distributed system that offers low latency access to data, we must protect applications from node failures (caused by software bugs and hardware/network failures). High availability of data and transparent handling of failed operations therefore become an important requirement for SnappyData.

R4. Consistency — As a highly available system that offers concurrent data access, it becomes important to ensure that all applications have a consistent view of data.

After an overview of our cluster architecture in section 5.1, we explain how SnappyData meets each of these requirements in the subsequent sections.

5.1 SnappyData Cluster Architecture

A SnappyData cluster is a peer-to-peer (P2P) network comprised of three distinct types of members (see figure 4).

1. Locator. Locator members provide discovery service for the cluster. They inform a new member joining the group about other existing members. A cluster usually has more than one locator for high availability reasons.

2. Lead Node. The lead node member acts as a Spark driver by maintaining a singleton `SparkContext`. There is one primary lead node at any given instance but there can be multiple secondary lead node instances on standby for fault tolerance. The lead node hosts a REST server to accept and run applications. The lead node also executes SQL queries routed to it by "data server" members.

3. Data Servers. A data server member hosts data, embeds a Spark executor, and also contains a SQL engine capable of executing certain queries independently and more efficiently than Spark. Data servers use intelligent query routing to either execute the query directly on the node, or pass it to the lead node for execution by Spark SQL.

5.2 High Concurrency in SnappyData

Thousands of concurrent ODBC and JDBC clients can simultaneously connect to a SnappyData cluster. To support this degree of concurrency, SnappyData categorizes incoming requests from these clients into (i) low latency requests and (ii) high latency ones. For low latency operations, we completely bypass Spark's scheduling mechanism and directly operate on the data. We route high latency operations (e.g., compute intensive queries) through Spark's fair scheduling mechanism. This makes SnappyData a responsive system, capable of handling multiple low latency short operations as well as complex queries that iterate over large datasets simultaneously.

5.3 State Sharing in SnappyData

A SnappyData cluster is designed to be a long running clustered database. State is managed in tables that can be shared across any number of connecting applications. Data is stored in memory and replicated to at least one other node in the system. Data can be persisted to disk in shared nothing disk files for quick recovery. (See section 4 for more details on table types and redundancy.) Nodes in the cluster stay up for a long time and their life-cycle is independent of application lifetimes. SnappyData achieves this goal by decoupling its process startup and shutdown mechanisms from those used by Spark.

5.4 High Availability in SnappyData

To explain SnappyData's approach to high availability, we first need to describe our underlying group membership service as our building block for providing high availability.

5.4.1 P2P Dynamic Group Membership Service

A Spark cluster uses a master-slave model, where slaves become aware of each other through a single master. Ensuring consistency between the slaves is coordinated through the master. For instance, in Spark, to broadcast a dataset and cache it on all executors, one has to first send the dataset to the driver node, which in turn replicates the data to each worker node. This is a reasonable strategy for small immutable datasets where the driver is rarely used. In contrast, SnappyData relies on a P2P connected system with an underlying active group membership system that ensures consistency between replicas. Strict membership management is a pre-requisite for managing the metadata governing the distributed consistency of data in the cluster. It allows SnappyData to offer lower latency guarantees even while faced with failure conditions. Next, we explain the building blocks of this group membership service (inherited from GemFire).

Discovery service — The discovery service's primary responsibility is to provide an initial list of known members, including all lead nodes and data servers.

Group coordination — The oldest member in the group automatically becomes the group coordinator. A group coordinator establishes a consistent view of the current membership of the system and ensures that this view is consistently known to all members.

Any new member first discovers the initial membership and the coordinator through the discovery service. All JOIN requests are received by the coordinator who confirms and informs everyone about the new member. When a new member joins, it may host a replica of some existing dataset. The coordinator also ensures virtual synchrony to ensure that no in-flight events are missed by the new member. All members establishe a direct communication channel with each other.

Failure handling — While failures are easy to detect when a socket endpoint fails (e.g., a node fails or the process dies), it is rather difficult to detect network partitions in a timely manner. To handle failures, we use multiple failure detection schemes, e.g., UDP neighbor ping and TCP channel. When any member detects a lack of response from another member, it sends a SUSPECT notification to the coordinator, which in turn perform a SUSPECT verification sequence to ensure the SUSPECT is indeed unreachable. If so, it it establishes a new membership view, distributes it to all members, and finally confirms the failure with the member that raised the suspicion.

5.4.2 *Achieving High Availability (HA)*

The group membership system described above plays a vital role in achieving HA.

- **Lead node HA:** Multiple lead nodes go through an election protocol to elect a primary. To accomplish this, we rely a distributed lock service (DLS) built using the group membership service. Only the first member who acquires the lock proceeds to become the lead node. Other lead nodes operate in a standby mode and go through the election protocol again if the primary fails.
- **Executor HA:** While Spark executors run within data servers, we have to ensure that the Spark driver can reschedule tasks on other executors as well. To allow this, we use the same leader election protocol as described above.

5.5 Transactional Consistency in SnappyData

SnappyData supports "read committed" and "repeatable read" transaction isolation levels. A transaction can be initiated using JDBC or ODBC using a single connection (transactions cannot span connections). Transactions are always coordinated on a single member (typically the first member to receive a write) and sub-coordinators are started on other nodes involved in the transaction. The transactional state itself is managed in an in-memory buffer on each node until the commit phase. We acquire write locks on all cohorts (replicas) as and when the write occurs. Our model assumes few or no conflicts and fails fast if the exclusive write lock cannot be obtained, in which case a write-write conflict exception is returned to the caller. Essentially, the design is tilted in favor of no centralized locking schema for scalability but assumes short-lived transactions with a small write set.

Given that all conflicts are resolved before the commit phase, the commit sequence involves a single commit message to all cohorts. To ensure atomic commits, the messaging is deeply integrated with the group consensus protocol built into the membership sub-system. The details of how consensus is established is beyond the scope of this paper. However, if any of the members fail to respond to the commit message, the group membership system will determine if the member is unreachable or is indeed dead and will remove the offending member from the distributed system. When the failed member recovers, it sheds its local state and recovers a consistent copy from another replica.

6. APPROXIMATION FOR INTERACTIVE AND STREAMING ANALYTICS

As mentioned in section 1, achieving interactive response times is a challenging task even when the data is kept in memory. In fact, any OLAP query that requires distributed shuffling of the records can take tens of seconds to minutes. Moreover, distributed clusters are often shared by hundreds of users and applications concurrently running such queries. Finally, bursty arrivals of high velocity streams can easily exceed the available resources, in which case queues will build up and latencies increase without bound [15].

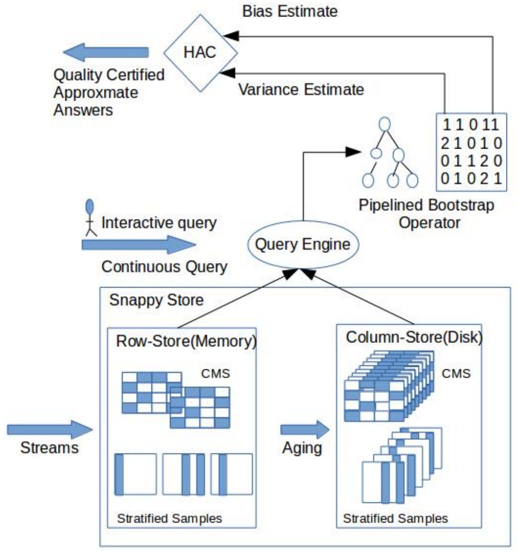

Figure 5: Approximate query processing in SnappyData

To ensure interactive response times under all these conditions, SnappyData's query engine is equipped with state-of-the-art AQP (approximate query processing) techniques. While traditional stream processors similarly resort to *load shedding*, they only provide accuracy guarantees for simple classes of SQL queries [15, 21, 29, 34]. To the best of our knowledge, SnappyData is the first to provide accuracy guarantees for arbitrarily complex OLAP queries on data streams. Figure 5 shows our AQP pipeline, which is explained next.

DDL for Approximation — SnappyData extends the DDL to allow users to include their approximation preference in their table or stream definitions. SnappyData uses this information to build appropriate forms of synopses (i.e., probabilistic data structures). Currently, users can specify any number of column sets to built a count-min sketch (CMS), a uniform sample, or a stratified sample on.[1] A CMS [27] allows for efficient top-K queries (a.k.a. heavy hitters), while a stratified sample [10, 18, 23] enables fast answers for queries with selective WHERE conditions on the stratified columns. In the example below, the user is specifying that queries will commonly have `zip_houseId` and `timestamp` in WHERE conditions, and thus need to be stratified on. In addition, top-K queries on `value` for a `zip_houseId` will be common.

```
CREATE TABLE meter_readings (
  property INT,
  timestamp INT,
  value DOUBLE,
  zip_houseId VARCHAR(20),
) USING column;

CREATE SAMPLED TABLE meter_readings_sampled_zip
OPTIONS (BASETABLE 'meter_readings'
              QCS 'zip_houseId,timestamp');

CREATE TOPK meter_readings_topk_value
```

[1]We plan to fully automate this process using the CliffGuard framework (http://cliffguard.org) to handle situations where past queries are not representative of future ones.

```
OPTIONS (BASETABLE 'meter_readings'
         KEY  'zip_houseId',
         AGGREGATE  'value');
```

Listing 5: Approximation DDL

Online Synopsis Maintenance and Aging — As streams are ingested, all relevant synopses are updated incrementally, using the Hokusai algorithm [27] for CMS and reservoir sampling for uniform and stratified samples. For synopses built on a stream, time is automatically added as another dimension to the set of user-specified columns. The time dimension allows SnappyData to continuously age the tail of the CMS matrix or sampled tuples into our compressed column-store format, while maintaining the last window (specified by application) in our in-memory row-store. Our current solution for join queries between large tables and streams is to include the join key in at least one of the stratified samples. We also plan to automatically include join-synopses [8] for foreign-key relationships in the schema.

Query Evaluation — Our approximate query engine automatically detects top-K queries and routes them to our CMS evaluation module. In the absence of an appropriate CMS, or when the resulting error does not meet user's accuracy requirements, the query is matched with a stratified sample whose column set best matches that if the query's WHERE clause. In the absence of a proper stratified sample, uniform samples are used as a last resort. When user's accuracy cannot be met with available synopses, appropriate action is taken depending on the High-level Accuracy Contract requested by the user (see below).

Pipelined Bootstrap Operator — To quantify our sampling error, we use bootstrap which can support almost arbitrary OLAP queries. We use Poissionized bootstrap [9], which annotates each tuple with 100–200 integers independently drawn from a Poisson(1) distribution. These integers succinctly represent the multiplicities of each tuple in each of the bootstrap replica. A special operator, called *pipelined bootstrap operator*, uses these multiplicities as tuples are pipelined through the physical plan to produce an empirical distribution of the approximate answers.

While all previous AQP engines have used bootstrap only to estimate confidence intervals, assuming that bias is negligible or that users themselves provide unbiased estimators [9, 25, 33, 43, 44], SnappyData uses bootstrap's empirical distribution to also estimate and correct the bias introduced during the approximation. (See [20, 28] for a description of bias correction using bootstrap.)

High-level Accuracy Contract (HAC) — In general, for a SQL query with m aggregate columns in its `SELECT` clause, each output row has $m + 1$ error terms: one to capture the row's probability of existence, and m terms for the errors of its aggregate columns.

Consequently, AQP solutions have historically faced two adoption barriers in practice: (i) appending error estimates to the query output might break the internal logic of existing BI (business intelligence) tools, and (ii) a typical database user will simply find a large number of errors associated with each row overwhelming.

To the best of our knowledge, SnappyData is the first to address these challenges through the use of a *High-level Accuracy Contract (HAC)* [28]. A HAC is a single number ϕ, where $0 \leq \phi \leq 1$, chosen by the end user. Given a particular ϕ, SnappyData guarantees that any results returned to users or BI tools will be at least $\phi \times 100\%$ accurate, in the following sense. Every output tuple whose probability of existence is below ϕ is omitted. However, aggregate values that do not meet the requested HAC will be dealt with by using one of the following policies (chosen by the user):

P1: *Do nothing.* All aggregate values are returned (possibly with a warning).

P2: *Use special symbols.* Aggregate values that do not meet the required HAC are replaced with special values (`NULL` or pre-defined values).

P3: *Drop the row.* The entire row is omitted if any of its aggregate columns do not meet the required HAC.

P4: *Fail.* The entire output relation is omitted, and a SQL exception is thrown, if any of the aggregate columns in any of the rows do not meet the required HAC.

This approach will allow users to control the system's behavior, without having to include the error columns in the output, and thus, without breaking the BI tools. Here, *do nothing* is the most lenient policy and *fail* is the strictest one. In the latter, the user can decide whether to re-run the query with a more lenient policy, or simply resort to exact query evaluation. The *drop the row* policy can affect the internal logic of the BI tools if it relies on the output's cardinality.

On the other hand, advanced users can explicitly request detailed error statistics through designated functions, shown in Listing 6.

```
SELECT callTowerId, avg(droppedPackets) AS fault
FROM CallDetailRecords
WHERE fault > 0.08
    AND existence_probability() > 0.95
    AND relative_error(satisfaction, 0.95)<0.1
GROUP BY callTowerId
```

Listing 6: While HAC shields the user from detailed statistics, they can still be requested explicitly

The HAC approach allows practitioners and end users to express their required level of accuracy in an intuitive fashion—as a single percentage—and without being overwhelmed with numerous statistics. It also provides a range of intuitive policies to cater to different levels of accuracy concerns, while still offering advanced users the ability to access and use detailed error statistics.

7. OTHER OPTIMIZATIONS

In this section, we present a few notable optimizations offered by SnappyData.

7.1 Locality-Aware Partition Design

One 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) expensive distributed locks across nodes to ensure transactional consistency [22, 42]. Besides the network costs, shuffling can also cause CPU bottlenecks by incurring excessive copying (between kernel and user space) and serialization costs [31]. To reduce the need for shuffling and distributed locks, we promote two fundamental ideas in our data model:

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

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 explicitly request that these tables be replicated. While most distributed data systems support co-partitioning, replicating data sets to all partitions to optimize joins is far less common. In SnappyData, when nodes join/leave, we ensure the replicas are maintained consistently in the presence of many in-flight updates in the distributed system.

7.2 Unified Memory Manager

SnappyData leverages Spark SQL for its columnar storage. When data is stored in column tables, it is managed as blocks or rows. Below, we describe how we integrate the memory manager of Spark with that of GemFire.

The memory manager in Spark divides the heap for use by different components with a cap on the total heap that can be safely allocated (90% by default). Each component (e.g., object cache, shuffle, unroll) is configured to use a separate fraction of the heap. If the available heap for a component is exhausted, then new allocations overflow to disk or fail. The accounting for the memory usage is done by a `BlockManager`.

GemFire attempts to provide the user with fine grained control over the memory used for tables. These controls are split into two categories.

At the process level, when the total heap usage exceeds a certain percentage, tables that are configured to evict will either overflow items to disk or eliminate them altogether (when the data in memory is used as a cache). At the table level, when the table exceeds a pre-configured entry count or memory size, entries are evicted to disk or destroyed. The eviction uses an LRU algorithm, ensuring that the most stale items are removed leaving more operationally used items in memory. One important difference from Spark's approach is that the heap monitoring is done by observing the actual heap usage as provided by the JDK's management interface for memory pools. Thus, all memory allocations in the JVM are accounted for in the decision making process. However, only tables and the runtime components mentioned above can evict data. A maximum cap on heap usage is also configured (90% by default) beyond which memory requests fail until adequate available memory has been restored in the process through the eviction process. In essence, by continuously monitoring the heap, GemFire aggressively prevents an Out-of-memory condition from occurring.

In the unified model, we apply the same thresholds for all spark managed memory also. i.e. Spark block manager starts to evict or overflow to disk when the eviction threshold is breached. And, similarly, spark block allocations will fail if the critical threshold is breached. This change does not change how Spark's need to allocate different fractions for different components.

Both GemFire and Spark also support offheap storage that currently needs to be configured separately. The data store in Spark can use offheap using Tachyon integration [5] [3] while runtime can use memory allocated outside of heap using JVM's private `sun.misc.Unsafe` API. Tables in GemFire can be configured to use offheap that uses the same *unsafe* API.

8. EXPERIMENTS

The main advantage offered by SnappyData is the reduced TCO by offering an integrated solution to replace the disparate environments used for streaming, OLTP and OLAP workloads. Since the long-term value of reduced operational costs and ease-of-use cannot be easily quantified, in this section we answer an alternative question: does SnappyData's hybrid solution come at the cost of a lower performance compared to highly specialized systems for OLAP, OLTP and stream processing?

To answer this question, we compared (i) SnappyData's OLAP performance against Spark SQL 1.5 using TPC-H benchmark, (ii) its OLTP performance against MemSQL using YCSB benchmark, and (iii) its approximate and streaming performance against exact stream processing using Twitter's live feed. Surprisingly, not only was SnappyData comparable to these highly specialized systems, in many cases it was considerably superior too.

Unless specified otherwise, in our experiments we used 7 machines with 32 cores and 64 GB RAM running Red Hat Enterprise Linux Server release 6.5.

8.1 OLAP Workload: TPC-H

TPC-H is a popular OLAP benchmark with 22 query types. SnappyData's DDL was used to create tables which were hash partitioned and colocated. Spark SQL API was used to create tables and load the data using Spark's caching API. We experimented with 1GB, 10GB and 100 GB datasets (a.k.a. 1x, 10x, and 100x scales, respectively). However, a known bug in Spark SQL 1.5 prevented it from running on the 100 GB dataset. The bug has been fixed in 1.6, but the official release was still in the works when we ran these tests.

We had to rewrite some of the queries so that they could be executed in Spark SQL. Given that Spark SQL is still evolving, we expect that it will soon improve to handle complex nested queries without the need for modifying them. For fairness, we used the modified queries for both Spark SQL and SnappyData. We cached tables for both products, ran each query 3 times, and recorded the average of the last two runs. In addition, while SnappyData supports the use of indexes (which would be beneficial for several queries), we opted not to create any indexes.

The results are shown in Figure 6. In summary, our experiments indicated that queries executed faster in SnappyData, particularly for those with one or more joins. On average, queries ran 73% and 52% faster on SnappyData, for the 1GB and 10GB datasets, respectively.

The reason behind SnappyData's superiority is that it models tables as partitioned or replicated (see section 4), and uses a number of optimizations for colocating them accordingly. We modeled the partitioned tables to be colocated and modeled all dimension tables to be replicated. For joins on colocated tables, SnappyData alters the query plan to avoid shuffling altogether because related items are already on the same node. In contrast, Spark SQL chooses an ex-

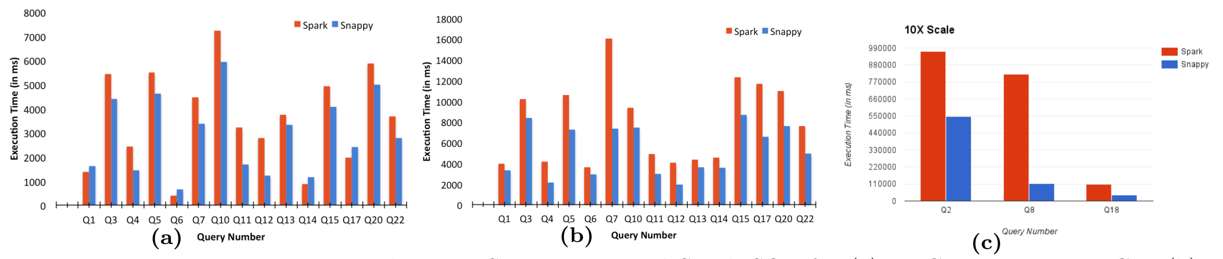

Figure 6: Response time comparisons between SnappyData and Spark SQL for (a) TPC-H queries on 1GB, (b) TPC-H queries on 10GB, and (c) Q2, Q8 and Q18 on 10GB

pensive shuffling plan. In general, SnappyData can optimize joins between the following table types.

	Column	Replicated	Partitioned
Column	√	√	×
Replicated	√	√	√
Partitioned	×	√	√

Table 1: Table types with join optimization in SnappyData

8.2 OLTP Workload: YCSB

We used Yahoo's Cloud Serving Benchmark (YCSB [19]) for emulating an OLTP workload, and compared SnappyData against MemSQL, as a state-of-the-art in-memory OLTP/OLAP commercial database.

For this experiment, we used YCSB's A, B, C and F workloads. In YCSB, each workload represents a particular mix of reads/writes, data sizes, and request distributions, and can be used to evaluate systems at different points in the performance space (see Table 2). We used 100 million records with the default redundancy for both SnappyData and MemSQL. Thus, the total data volume was 100 GB for both systems.

As shown in figure 7a, on average, SnappyData delivered 51% higher throughput across all A, B, C, and F workloads. SnappyData also achieved remarkably lower latencies (43%) compared to MemSQL across all workloads (see figure 7b).

8.3 AQP and Stream Analytics: Twitter

To study the effectiveness of our AQP in enabling interactive analytics over large volumes of streaming data, we compared two alternatives: (i) running the continuous query on the entire stream to provide exact answers, and (ii) running the query on a 2% stratified sample (i.e., AQP) to provide an approximate answer.

We used Twitter's live feed to capture 130+ million tweets using Spark's DataSource API, placed in Kafka queues and ingested into SnappyData. We ran a continuous query to report the top 10 hashtags at regular intervals, and recorded the execution time difference between the exact and approximate query.

The results are shown in figure 7c for different window sizes. For the smallest window size (0.5 min), SnappyData delivered a highly accurate approximate answer 3x faster. This performance gap rapidly grew with the window size, reaching 20x for a 12-minute window. As shown in figure 7c, for this window size the execution time of exact query was 19 seconds, which is hardly an interactive speed, while our stratified sampling strategy maintained a consistent performance.

For all window sizes, the order of hashtags were mostly consistent with the exact results. Even the actual counts,

Workload	Operations
Update heavy(A)	Read 50% Update 50%
Read heavy(B)	Read 95% Update 5%
Read only(C)	Read 100%
Readmodifywrite(F)	Read 50% Read-Modify-Write 50%

Table 2: Workload Operations Table

showed 90-95% accuracy in our tests, which could be further improved by using a larger sampling rate.

9. RELATED WORK

Stream processing — There are numerous commercial solutions for stream and complex event processing, such as Samza [1], Storm [38], Aurora/Tibco Streambase, Google's MillWheel [12], Confluent, sqlstream [4], and Spark Streaming [41]. (For academic solutions see [30, 36] and the references within.) While these systems support real-time monitoring and continuous queries and can handle bursty arrivals of data, they are generally not designed for scalable analytics the way that traditional OLAP databases are. While academic prototypes [13, 17] provide load shedding to cope with bursty arrivals, they only provide accuracy guarantees for simple aggregate queries [15, 21, 29] whereas SnappyData can provide streaming AQP for complex analytic queries.

There most related papers are DataCell [26], AIM [16], and Druid [40]. While there are many similarities in our goals and approach, AIM's design is focused on a telco-specific solution while we target a general-purpose operational DB with full transaction support. Similarly, DataCell provides no OLTP support and Druid does not offer SQL.

Transaction support — Both transactional DBMSs and modern key-value stores (e.g., HBase, Cassandra, MongoDB) are highly scalable for point reads and writes, they are not apt at OLAP-style analytics. A few commercial hybrid in-memory engines, such as MemSQL and SAP Hana, optimize for both OLTP and OLAP workloads. MemSQL lacks streaming support, and Hana's Smart Data Streaming is an add-on that can interface with Hana's engine, but is not sufficiently integrated to capitalize on Hana's OLAP-style optimizations.

Interactive SQL analytics — Both MPP (massively parallel processing) databases and modern SQL-on-Hadoop engines (e.g., Hive [37], Impala [24], and Spark SQL [14]) provide scalable OLAP analytics through various optimizations for table scans, group by-aggregations, joins.

AQP — Several AQP systems have used stratified samples [10, 11, 18, 23] and bootstrap-based error estimation [9, 25, 33, 44] for interactive analytics. However, to the best of our knowledge, SnappyData is the first to (i) use bootstrap for automatic bias correction, and (ii) provide high-level accuracy contracts to end users.

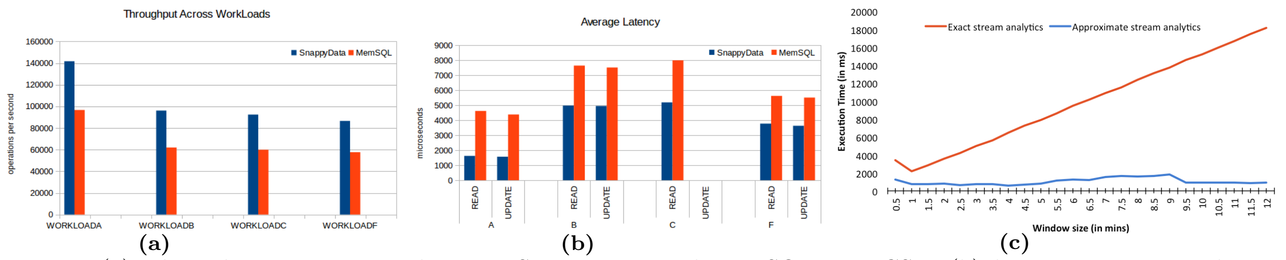

Figure 7: (a) Throughput comparison between SnappyData and MemSQL on YCSB, (b) latency comparison between SnappyData and MemSQL on YCSB, and (c) execution time difference for exact vs. approximate stream analytics on Twitter

10. CONCLUSION

In this paper, we proposed a unified platform for real time operational analytics, SnappyData, to support OLTP, OLAP, and stream analytics in a single integrated solution. We presented the approach that we have taken to deeply integrate Apache Spark (a computational engine for high throughput analytics) with GemFire (a scale out in-memory transactional store). SnappyData extends Spark SQL and Spark Streaming API with mutability semantics, and offers various optimizations to enable collocated processing of streams and stored datasets. We also made the case for integrating approximate query processing into this platform as a critical differentiator for supporting real time operational analytics over big stored and streaming data.

Finally, we evaluated the performance of our integrated solution using popular benchmarks. We believe that our platform significantly lowers the TCO for operational real-time analytics by combining products that would otherwise have to be managed, deployed, and monitored separately.

Bibliography

[1] Apache Samza. http://samza.apache.org/.
[2] IBM InfoSphere BigInsights. http://tinyurl.com/ouphdss.
[3] Spark RDD Persistence. http://tinyurl.com/pw8dq3q.
[4] sqlstream. http://www.sqlstream.com/.
[5] Tachyon Project. http://tachyon-project.org.
[6] TIBCO StreamBase. http://www.streambase.com/.
[7] D. Abadi et al. *The Design and Implementation of Modern Column-Oriented Database Systems*. 2013.
[8] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.
[9] 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.
[10] 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.
[11] 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.
[12] T. Akidau et al. MillWheel: fault-tolerant stream processing at internet scale. *PVLDB*, 2013.
[13] A. Arasu et al. Stream: the stanford stream data manager. In *SIGMOD*, 2003.
[14] M. Armbrust et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.
[15] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, 2004.
[16] L. Braun et al. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD*, 2015.
[17] S. Chandrasekaran et al. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.
[18] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.
[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.
[20] B. Efron and R. Tibshirani. *An introduction to the bootstrap*, volume 57. CRC press, 1993.
[21] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. MobiQual: QoS-aware Load Shedding in Mobile CQ Systems. In *ICDE*, 2008.
[22] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 2007.
[23] S. Joshi and C. Jermaine. Robust Stratified Sampling Plans for Low Selectivity Queries. In *ICDE*, 2008.
[24] M. Kornacker et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.
[25] N. Laptev et al. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 2012.
[26] E. Liarou et al. Monetdb/datacell: online analytics in a streaming column-store. *PVLDB*, 2012.
[27] S. Matusevych, A. Smola, and A. Ahmed. Hokusai-sketching streams in real time. *arXiv preprint arXiv:1210.4891*, 2012.
[28] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Engineering Bulletin*, 2015.
[29] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, 2010.
[30] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. In *SIGMOD*, 2012.
[31] K. Ousterhout et al. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.
[32] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. *CoRR*, 2015.
[33] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *SIGMOD*, 2005.
[34] N. Tatbul et al. Load shedding in a data stream manager. In *VLDB*, 2003.
[35] M. Telecom. GPS trackers trial may help people with dementia. http://tinyurl.com/zphr6au.
[36] 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.
[37] A. Thusoo et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2009.
[38] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.
[39] R. Xin and J. Rosen. Project Tungsten: Bringing Spark closer to bare metal. http://tinyurl.com/mzw7hew.
[40] F. Yang et al. Druid: a real-time analytical data store. In *SIGMOD*, 2014.
[41] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.
[42] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.
[43] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. Abs: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.
[44] 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.