SnappyData: A Hybrid Transactional Analytical Store Built On Spark*

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

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

[1] {jramnarayan,barzan,swale,smenon,nkumar,hbhanawat}@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).

In this demonstration, after presenting a few use case scenarios, we exhibit SnappyData as our our in-memory solution for delivering truly interactive analytics (i.e., a couple of seconds), when faced with large data volumes or high velocity streams. We show that SnappyData can exploit state-of-the-art approximate query processing techniques and a variety of data synopses. Finally, we allow the audience to define various *high-level accuracy contracts (HAC)*, to communicate their accuracy requirements with SnappyData in an intuitive fashion.

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.

SQL-on-Hadoop solutions (e.g., Hive, Impala, and Spark SQL) 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 need to be used alongside a streaming engine (e.g., Storm [14], Kafka, Confluent) to support streaming processing.

Finally, stream processors (e.g., Samza [1]) provide some form of state management, but only allow support simple analytics over data streams. This is because complex analytics require the same optimizations used in a OLAP engine [7, 10], such as columnar formats and efficient operators for joining, grouping, or aggregating large histories. For example, many of our customers in Industrial IoT derive insight by 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 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) or a homegrown application, writing state to a key-value store (e.g., Cassandra) or an in-memory data

*For a detailed paper, see [13].

SIGMOD/PODS'16 June 26 - July 01, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2899408

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 — *We aim to deliver interactive-speed analytics with modest investments in cluster infrastructure and far less complexity than today.* SnappyData fulfills 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. When Spark executes tasks in a partitioned manner, it keeps all 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. To support high concurrency, we also extend the runtime with a "Job Server" that decouples applications from data servers, while sharing state across many clients and applications.

(b) **Unified API for OLAP, OLTP, and streaming.** We extend Spark's 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.

(c) **Optimized Spark applications** Our deeply integrated store eliminates the need for an external one (e.g., a KV store) for Spark applications. This improves overall performance by minimizing network traffic and serialization costs. In addition, by promoting colocated schema designs, in many scenarios SnappyData eliminates the need for shuffling altogether.

(d) To deliver analytics at truly interactive speeds, we have equipped SnappyData with state-of-the-art AQP techniques [4, 5] as well as several novel features. Unlike previous AQP engines [2, 3, 6, 9, 15], Instead of overwhelming end users with numerous statistics, we provide an intuitive means for expressing accuracy requirements as *high-level accuracy contracts (HAC)* [11]. SnappyData is also the first AQP engine to offer automatic bias correction for arbitrarily complex SQL queries. Finally, while traditional load shedding techniques are restricted to simple queries [12], SnappyData provides error estimates for arbitrarily complex queries on streams.

2. SYSTEM OVERVIEW

2.1 System Architecture

Figure 1 depicts the core components of SnappyData, where Spark's original components are highlighted in gray. To simplify, we have omitted standard components, such as security and monitoring.

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.

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. Our stream processing is primarily through Spark Streaming, but it is integrated and runs *in-situ* with our store.

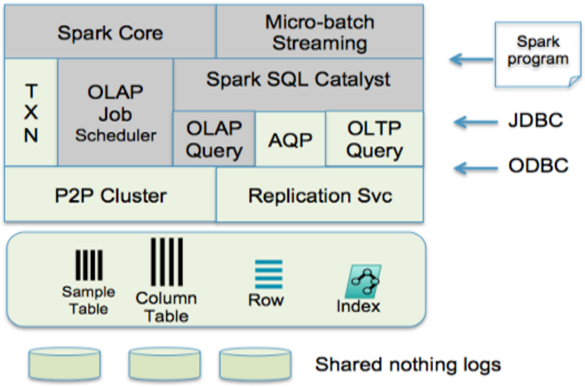


Figure 1: SnappyData's core components

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.

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.

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.

2.2 Data Ingestion Pipeline

SnappyData focuses on use cases that involve 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 (re-

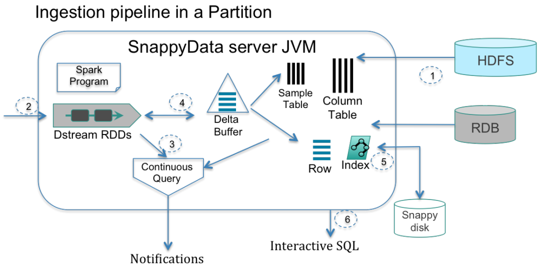


Figure 2: Data ingestion pipeline in SnappyData

ferred 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 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. This delta row buffer is merged 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.

3. DEMONSTRATION DETAILS

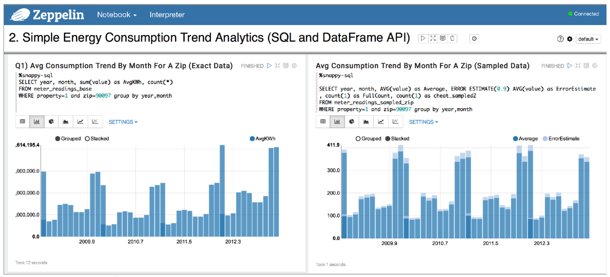
Our demonstration scenario imagines the audience as users of a energy management system, monitoring in real time, metrics from thousands to millions of smart meters to understand load distribution, outliers, trends and so on. We organize our demonstration into two phases:

1. A brief introduction to the main system functionalities, in which we will exhibit the key components and features of our system.

2. A hands-on phase where the audience is invited to directly interact with the system and test its capabilities.

Our primary goal with this demonstration is to decisively show that curated sampled data sets and synopsis data structures can be extremely effective in providing quick responses to arbitrary analytic class queries at interactive speeds. We show side by side comparison to native Spark in-memory performance.

For interactive analytics on historical data, we store meter readings for households and businesses for the last few years and permit our demonstration users to try out queries using Apache Zeppelin—a web based notebook for interactive analytics (see Figure 3). They would be able to create our CMS (count-min-sketch [8]) based structures for managing time series meter readings across multiple dimensions or create one or more stratified samples.



2. Simple Energy Consumption Trend Analytics (SQL and DataFrame API)

Q1) Avg Consumption Trend By Month For A Zip (Exact Data) FINISHED

```
%snappy-sql
SELECT year, month, sum(value) as AvgKWh, count(*)
FROM meter_readings_base
WHERE property=1 and zip=90097 group by year,month
```

Took 12 seconds

Avg Consumption Trend By Month For A Zip (Sampled Data) FINISHED

```
%snappy-sql

SELECT year, month, AVG(value) as Average, ERROR ESTIMATE(0.9) AVG(value) as ErrorEstimate
, count(1) as FullCount, count(1) as cheat_sampled2
FROM meter_readings_sampled_zip
WHERE property=1 and zip=90097 group by year,month
```

Took 1 seconds

Figure 3: A screenshot of the demonstration scenario using Zeppelin and SnappyData

The demonstration is divided into 3 different notebooks. The first notebook introduces the APIs for sampling structures, loading data, and monitoring memory utilization. Moreover, the users will be given an opportunity to define their own HACs and observe their impact on the cardinality, accuracy, and response time of of various analytical queries.

The second notebook demonstrates the power of sampling and summary structures on already ingested data. The third notebook demonstrates the use of SQL-based "continuous queries" for detecting complex patterns in streaming data over arbitrary time windows. User will be able to choose various forms of exact or approximate answers using our CMS-based structures. For instance, energy utilization pattern over few mins correlated with similar periods in the past may predict future demand and enable outage management.

Bibliography

[1] Apache Samza. http://samza.apache.org/.

[2] Fast, approximate analysis of big data (yahoo's druid). http://yahooeng.tumblr.com/post/135390948446/data-sketches.

[3] Presto: Distributed SQL query engine for big data. https://prestodb.io/docs/current/release/release-0.61.html.

[4] 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.

[5] 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.

[6] 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.

[7] L. Braun et al. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD*, 2015.

[8] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55, 2005.

[9] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[10] E. Liarou et al. Monetdb/datacell: online analytics in a streaming column-store. *PVLDB*, 2012.

[11] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Engineering Bulletin*, 2015.

[12] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, 2010.

[13] J. Ramnarayan, B. Mozafari, S. Menon, et al. Snappydata: Streaming, transactions, and interactive analytics in a unified engine. http://web.eecs.umich.edu/ mozafari/php/data/uploads/snappy.pdf, 2016.

[14] A. Toshniwal et al. Storm@twitter. In *SIGMOD*, 2014.

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