SnappyData

Barzan Mozafari
University of Michigan, Ann Arbor, MI, USA

Introduction

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

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

Hybrid transaction/analytical processing (HTAP) systems, such as `MemSQL`, support both OLTP and OLAP queries by storing data in dual formats (row and columns) but need to be used alongside an external streaming engine (e.g., `Storm` (Toshniwal et al. 2014), `Kafka`, `Confluent`) to support stream processing. They also lack approximation features required for interactive-speed analytics or visualization workloads (Park et al. 2016).

Finally, there are numerous academic (Chandrasekaran et al. 2003; Mozafari et al. 2012; Thakkar et al. 2011) and commercial (Apache Samza; Toshniwal et al. 2014; TIBCO; Akidau et al. 2013) solutions for stream and event processing. Although some stream processors provide some form of state management or transactions (e.g., `Samza` (Apache Samza), `Liquid` (Fernandez et al. 2015), `S-Store` (Meehan et al. 2015)), they only allow simple queries on streams. However, more complex analytics, such as joining a stream with a large history table, need the same optimizations used in an OLAP engine (Liarou et al. 2012; Braun et al. 2015; Thakkar et al. 2011). For example, streams in IoT are continuously ingested and correlated with large historical data. Trill (Chandramouli et al. 2014) supports diverse analytics on streams and columnar data but lacks transactions.

This article is based on Mozafari et al. (2017), authored by Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, Kishor Bachhav

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_258-1

DataFlow (Akidau et al. 2015) focuses on logical abstractions rather than a unified query engine.

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

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

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

D1. Increased complexity and total cost of ownership: The use of incompatible and autonomous systems significantly increases their total cost of ownership. Developers have to master disparate APIs, data models, and tuning options for multiple products. Once in production, operational management is also a nightmare. To diagnose the root cause of a problem, highly paid experts spend hours to correlate error logs across different products.

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

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

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

Challenges

`SnappyData`’s goal is to reduce complexity and improve performance by offering streaming, transaction processing, and interactive analytics in a single cluster (Ramnarayan et al. 2016). Realizing this goal involves overcoming several challenges. The first challenge is the drastically different data structures and query processing paradigms that are optimal for each type of workload. For example, column stores are optimal for analytics, transactions need write-optimized row-stores, and infinite streams are best handled by sketches and windowed data structures. Likewise, while analytics thrive with batch processing, transactions rely on point lookups/updates, and streaming engines use delta/incremental query processing. Marrying these conflicting mechanisms in a single system is challenging, as is abstracting away this heterogeneity from programmers.

Another challenge is the difference in expectations of high availability (HA) across different workloads. Scheduling and resource provisioning are also harder in a mixed workload of streaming jobs, long-running analytics, and short-lived transactions. Finally, achieving interactive analytics becomes nontrivial when deriving insight requires joining a stream against large historical data (Makin' Bacon and the Three Main Classes of IoT Analytics).

Approach Overview

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

Through a common set of abstractions, Spark allows programmers to tackle a confluence of different paradigms (e.g., streaming, machine learning, SQL analytics). Spark's core abstraction, a Resilient Distributed Dataset (RDD), provides fault tolerance by efficiently storing the lineage of all transformations instead of the data. The data itself is partitioned across nodes and if any partition is lost, it can be reconstructed using its lineage. The benefit of this approach is twofold: avoiding replication over the network and higher throughput by operating on data as a batch. While this approach provides efficiency and fault tolerance, it also requires that an RDD be immutable. In other words, Spark is simply designed as a computational framework and therefore (i) does not have its own storage engine and (ii) does not support mutability semantics. (Although IndexedRDD (Indexedrdd for apache spark) offers an updatable key-value store (Indexedrdd for apache spark), it does not support colocation for high-rate ingestions or distributed transactions. It is also unsuitable for HA, as it relies on disk-based checkpoints for fault tolerance.)

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

Best of two worlds – To combine the best of both worlds, SnappyData seamlessly integrates Spark and GemFire runtimes, adopting Spark as the programming model with extensions to support mutability and HA (high availability) through GemFire's replication and fine-grained updates. This marriage, however, poses several nontrivial challenges. For instance, when ingesting a stream, SnappyData processes the incoming stream as a batch, avoids replication, and replays from the source on a failure. To avoid a tuple-at-a-time replication, the processed state can be written to the store in batches. Recovery from failure will thus be limited to the time needed to replay a single batch.

Architecture

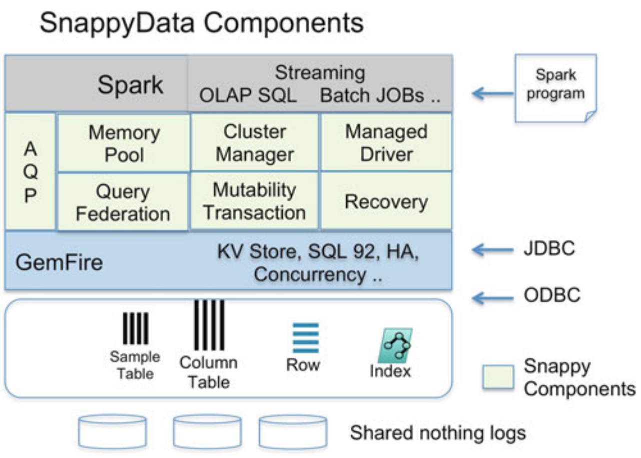
Figure 1 depicts SnappyData's core components (the original components from Spark and GemFire are highlighted).

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

SnappyData supports two programming models – SQL (by extending SparkSQL dialect) and Spark's API. Thus, one can perceive SnappyData as a SQL database that uses Spark's API as its language for stored procedures. Stream processing in SnappyData is primarily through Spark Streaming, but it is modified to run in situ with SnappyData's store.

SQL queries are federated between Spark's Catalyst and GemFire's OLTP engine. An initial query plan determines if the query is a

SnappyData, Fig. 1 SnappyData's core components



low-latency operation (e.g., a key-based lookup) or a high-latency one (scans/aggregations). SnappyData avoids scheduling overheads for OLTP operations by immediately routing them to appropriate data partitions.

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

A Unified API

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

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

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

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

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

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

Listing .1 Working with DataFrames in SnappyData

`SnappyContext`, which is an extension of `SQLContext`. Listing .1 is an example of using `SnappyContext`.

Stream processing often involves maintaining counters or more complex multidimensional summaries. As a result, stream processors today are either used alongside a scale-out in-memory key-value store (e.g., Storm with Redis or Cassandra) or come with their own basic form of state management (e.g., Samza, Liquid (Fernandez et al. 2015)). These patterns are often implemented in the application code using simple get/put APIs. While these solutions scale well, most users tend to modify their search patterns and trigger rules quite often. These modifications require expensive code changes and lead to brittle and hard-to-maintain applications.

In contrast, SQL-based stream processors offer a higher-level abstraction to work with streams but primarily depend on row-oriented stores (e.g., IBM; TIBCO; Meehan et al. (2015)) and are thus limited in supporting complex analytics. To support continuous queries with scans, aggregations, top-K queries, and joins with historical and reference data, some of the same optimizations found in OLAP engines must be incorporated in the streaming engine (Liarou et al. 2012). Thus, `SnappyData` extends `Spark Streaming` to allow declaring and querying streams in SQL. More importantly, `SnappyData` provides OLAP-style optimizations to enable scalable stream analytics, including columnar formats, approximate query processing, and co-partitioning (SnappyData 2016).

Hybrid Store: Row and Column Tables

Tables can be partitioned or replicated and are primarily managed in memory with one or more consistent replicas. The data can be managed in Java heap memory or off-heap. Partitioned tables are always partitioned horizontally across the cluster. For large clusters, `SnappyData` allows data servers to belong to one or more logical groups, called "server groups." The storage format can be "row" (either partitioned or replicated tables) or "column" (only supported for partitioned tables) format. Row tables incur a higher in-memory footprint but are well suited to random updates and point lookups, especially with in-memory indexes. Column tables manage column data in contiguous blocks and are compressed using dictionary, run-length, or bit encoding (Xin and Rosen).

`SnappyData` extends Spark's column store to support mutability. Updating row tables is trivial. When records are written to column tables, they first arrive in a *delta row buffer* that is capable of high write rates and then age into a columnar form. The delta row buffer is merely a partitioned row table that uses the same partitioning strategy as its base column table. This buffer table is backed by a conflating queue that periodically empties itself as a new batch into the column table. Here, conflation means that consecutive updates to the same record result in only the final state getting transferred to the column store. For example, inserted/updated records followed by deletes are removed from the queue. The delta row buffer itself uses copy-on-write semantics to ensure that concurrent application updates do not cause inconsistency (Abadi et al. 2013). `SnappyData` extends `Spark`'s Catalyst optimizer to merge the delta row buffer during query execution.

Probabilistic Store

Achieving interactive response time is challenging when running complex analytics on streams, e.g., joining a stream with a large table (Mozafari and Zaniolo 2010). Even OLAP queries on stored datasets can take tens of seconds to complete if they require a distributed shuffling of records or if hundreds of concurrent queries run in the cluster (Agarwal et al. 2012). In such cases, `SnappyData`'s storage engine is capable of using probabilistic structures to dramatically reduce the volume of input data and provide approximate but extremely fast answers. In this regard, `SnappyData` can be seen as the first full-fledged commercial AQP engine (see Mozafari (2017) for why the adoption of AQP has not previously slowed). `SnappyData`'s probabilistic structures include uniform samples, stratified samples, and sketches (Mozafari and Niu 2015). Unlike `VerdictDB` (Park et al. 2018; He et al. 2018) and Database Learning (Park et al. 2017), which are agnostic of the underlying query engine, `SnappyData`'s probabilistic store is tightly integrated into its query processing logic. `SnappyData`'s approach is also different from other AQP engines (Zeng et al. 2014a; Agarwal et al. 2012; Zeng et al. 2014b), in the way that it creates and maintains these structures efficiently and in a distributed manner. However, given these structures, `SnappyData` uses off-the-shelf error estimation techniques (Agarwal et al. 2014). `SnappyData`'s sample selection and maintenance strategies are discussed next.

Sample selection – Unlike uniform samples, choosing which stratified samples to build is a nontrivial problem. The key question is which sets of columns to build a stratified sample on. Prior work has used skewness, popularity, and storage cost as the criteria for choosing column sets (Agarwal et al. 2012, 2013). SnappyData extends these criteria as follows: for any declared or foreign-key join, the join key is included in a stratified sample in at least one of the participating relations (tables or streams). However, SnappyData never includes a table's primary key in its stratified sample(s). Furthermore, SnappyData uses an open-source tool, called WorkloadMiner, which automatically analyzes past query logs and reports a rich set of statistics (CliffGuard). These statistics guide SnappyData's users through the sample selection process. WorkloadMiner is integrated into CliffGuard. CliffGuard guarantees a robust physical design (e.g., set of samples), which remains optimal even if future queries deviate from past ones (Mozafari et al. 2015).

Once a set of samples is chosen, the challenge is how to update them, which is a key differentiator between SnappyData and previous AQP systems that use stratified samples (Chaudhuri et al. 2007; Agarwal et al. 2013; Zeng et al. 2015).

Sample maintenance – Previous AQP engines that use offline sampling update and maintain their samples periodically using a single scan of the entire data (Mozafari and Niu 2015). This strategy is not suitable for SnappyData with streams and mutable tables for two reasons. First, maintaining per-stratum statistics across different nodes in the cluster is a complex process. Second, updating a sample in a streaming fashion requires maintaining a reservoir (Vitter et al. 1985; Al-Kateb and Lee 2010), which means the sample must either fit in memory or be evicted to disk. Keeping samples entirely in memory is impractical for infinite streams unless the sampling rate is perpetually decreased. Likewise, disk-based reservoirs are inefficient as they require retrieving and removing individual tuples from disk as new tuples are sampled.

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

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

State Sharing

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

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

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

Locality-Aware Partition Design

A major challenge in horizontally partitioned distributed databases is to restrict the number of nodes involved in order to minimize (i) shuffling during query execution and (ii) distributed locks (Helland 2007; Zamanian et al. 2015). In addition to network costs, shuffling can also cause CPU bottlenecks by incurring excessive copying (between kernel and user space) and serialization costs (Ousterhout et al. 2015). To reduce the need for shuffling and distributed locks, SnappyData's data model promotes two fundamental ideas:

1. **Co-partitioning with shared keys** – A common technique in data placement is to take the application's access patterns into account. SnappyData pursues a similar strategy: since joins require a shared key, it co-partitions related tables on the join key. SnappyData's query engine can then optimize its query execution by localizing joins and pruning unnecessary partitions.
2. **Locality through replication** – Star schemas are quite prevalent, wherein a few ever-growing fact tables are related to several dimension tables. Since dimension tables are relatively small and change less often, schema designers can ask SnappyData to replicate these tables. SnappyData particularly uses these replicated tables to optimize joins.

Dynamic rebalancing of data – When access is non-uniformly distributed across the keys, a load imbalance occurs where a few servers end up performing most of the work. For instance, when tracking users' browsing behavior on a website, a few popular pages will dominate the rest. SnappyData provides metrics on which nodes are being accessed heavily and also provides administrative APIs that can be used to move "hot buckets" of data to a different node. If the imbalance is a memory usage imbalance, admin APIs can be used to trigger a rebalance which is a non blocking operation that moves buckets of data to less loaded nodes in the background and restores memory balance. Used effectively, rebalancing prevents hotspots from developing in the system and avoid performance bottlenecks.

Hybrid Cluster Manager

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

High Availability

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

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

Failure recovery – Recovery in Spark is based on logging the transformations used to build an RDD (i.e., its lineage) rather than the actual data. If a partition of an RDD is lost, Spark has sufficient information to recompute just that partition (Zaharia et al. 2012). Spark can also checkpoint RDDs to stable storage to shorten the lineage, thereby shortening the recovery time. The decision of when to checkpoint, however, is left to the user. GemFire, on the other hand, relies on replication for instantaneous recovery but at the cost of lower throughput. SnappyData merges these recovery mechanisms as follows:

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

Hybrid Scheduler and Provisioning

Thousands of concurrent clients can simultaneously connect to a SnappyData cluster. To support this degree of concurrency, SnappyData categorizes incoming requests as low- and high-latency operations. By default, SnappyData treats a job as a low-latency operation unless it accesses a columnar table. However, applications can also explicitly label their latency sensitivity. SnappyData allows low-latency operations to bypass Spark's scheduler and directly operate on the data. High-latency operations are passed through Spark's fair scheduler. However, among the low-latency operations, SnappyData still relies on a simple FIFO policy (other systems, such as MariaDB or MySQL, use more sophisticated algorithms for transaction scheduling, e.g., VATS (Huang et al. 2017) or MySQL's CATS (Tian et al. 2018)). For low-latency operations, SnappyData attempts to reuse their executors to maximize their data locality (in-process). For high-latency jobs, SnappyData dynamically expands their compute resources while retaining the nodes caching their data.

Consistency Model

SnappyData relies on GemFire for its consistency model. GemFire supports "read committed" and "repeatable read" transaction isolation levels using a variant of the Paxos algorithm (Gray and Lamport 2006). Transactions detect write-write conflicts and assume that writers rarely conflict. When write locks cannot be obtained, transactions abort without blocking (Apache Geode).

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

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

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

Conclusion

SnappyData is a unified platform for real-time operational analytics, which supports OLTP, OLAP, and stream analytics in a single integrated solution. SnappyData's approach is a deep integration of a computational engine for high-throughput analytics (Spark) with a scale-out in-memory transactional store (GemFire). SnappyData extends SparkSQL and Spark Streaming APIs with mutability semantics and offers various optimizations to enable colocated processing of streams and stored datasets. SnappyData has integrated approximate query processing for enabling real-time operational analytics over large (stored or streaming) data. Overall, SnappyData's goal is to yield a significantly lower TCO for mixed workloads compared to using disparate products that are managed, deployed, and monitored separately.

References

Apache Geode. http://geode.incubator.apache.org/

Apache Samza. http://samza.apache.org/

CliffGuard. A general framework for robust and efficient database optimization. http://www.cliffguard.org

Exactly-once processing with trident – the fake truth. https://www.alooma.com/blog/trident-exactly-once

IBM. InfoSphere BigInsights. http://tinyurl.com/ouphdss

Indexedrdd for apache spark. https://github.com/amplab/spark-indexedrdd

Makin' Bacon and the Three Main Classes of IoT Analytics. http://tinyurl.com/zlc6den

TIBCO. StreamBase. http://www.streambase.com/

SnappyData (2016) Streaming, transactions, and interactive analytics in a unified engine. http://web.eecs.umich.edu/~mozafari/php/data/uploads/snappy.pdf

Abadi D et al (2013) The design and implementation of modern column-oriented database systems. Found Trends Databases 5(3):197–280

Agarwal S, Panda A, Mozafari B, Iyer AP, Madden S, Stoica I (2012) Blink and it's done: interactive queries on very large data. In: PVLDB

Agarwal S, Mozafari B, Panda A, Milner H, Madden S, Stoica I (2013) BlinkDB: queries with bounded errors and bounded response times on very large data. In: EuroSys

Agarwal S, Milner H, Kleiner A, Talwalkar A, Jordan M, Madden S, Mozafari B, Stoica I (2014) Knowing when you're wrong: building fast and reliable approximate query processing systems. In: SIGMOD

Akidau T et al (2013) MillWheel: fault-tolerant stream processing at internet scale. In: PVLDB

Akidau T et al (2015) The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: PVLDB

Al-Kateb M, Lee BS (2010) Stratified reservoir sampling over heterogeneous data streams. In: SSDBM

Barber R, Huras M, Lohman G, Mohan C, Mueller R, Özcan F, Pirahesh H, Raman V, Sidle R, Sidorkin O et al (2016) Wildfire: concurrent blazing data ingest and analytics. In: SIGMOD

Braun L et al (2015) Analytics in motion: high performance event-processing and real-time analytics in the same database. In: SIGMOD

Chandramouli B et al (2014) Trill: a high-performance incremental query processor for diverse analytics. In: PVLDB

Chandrasekaran S et al (2003) TelegraphCQ: continuous dataflow processing. In: SIGMOD

Chaudhuri S, Das G, Narasayya V (2007) Optimized stratified sampling for approximate query processing. ACM Trans Database Syst 32(2):9

Fernandez RC et al (2015) Liquid: unifying nearline and offline big data integration. In: CIDR

Gray J, Lamport L (2006) Consensus on transaction commit. ACM Trans Database Syst 31(1):133–160

He W, Park Y, Hanafi I, Yatvitskiy J, Mozafari B (2018) Demonstration of VerdictDB, the platform-independent AQP system. In: SIGMOD

Helland P (2007) Life beyond distributed transactions: an apostate's opinion. In: CIDR

Huang J, Mozafari B, Schoenebeck G, Wenisch T (2017) A top-down approach to achieving performance predictability in database systems. In: SIGMOD

Liarou E et al (2012) Monetdb/datacell: online analytics in a streaming column-store. In: PVLDB

Meehan J et al (2015) S-store: streaming meets transaction processing. In: PVLDB

Mozafari B (2017) Approximate query engines: commercial challenges and research opportunities. In: SIGMOD

Mozafari B, Zaniolo C (2010) Optimal load shedding with aggregates and mining queries. In: ICDE

Mozafari B, Niu N (2015) A handbook for building an approximate query engine. IEEE Data Eng Bull 38(3):3–29

Mozafari B, Zeng K, Zaniolo C (2012) High-performance complex event processing over xml streams. In: SIGMOD

Mozafari B, Ye Goh EZ, Yoon DY (2015) CliffGuard: a principled framework for finding robust database designs. In: SIGMOD

Mozafari B, Ramnarayan J, Menon S, Mahajan Y, Chakraborty S, Bhanawat H, Bachhav K (2017) SnappyData: a unified cluster for streaming, transactions, and interactive analytics. In: CIDR

Ousterhout K et al (2015) Making sense of performance in data analytics frameworks. In: NSDI

Park Y, Cafarella M, Mozafari B (2016) Visualization-aware sampling for very large databases. In: ICDE

Park Y, Tajik AS, Cafarella M, Mozafari B (2017) Database learning: towards a database that becomes smarter every time. In: SIGMOD

Park Y, Mozafari B, Sorenson J, Wang J (2018) VerdictDB: universalizing approximate query processing. In: SIGMOD

Ramnarayan J, Mozafari B, Menon S, Wale S, Kumar N, Bhanawat H, Chakraborty S, Mahajan Y, Mishra R, Bachhav K (2016) SnappyData: a hybrid transactional analytical store built on spark. In: SIGMOD

Thakkar H, Laptev N, Mousavi H, Mozafari B, Russo V, Zaniolo C (2011) SMM: a data stream management system for knowledge discovery. In: ICDE

Tian B, Huang J, Mozafari B, Schoenebeck G, Wenisch T (2018) Contention-aware lock scheduling for transactional databases. In: PVLDB

Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, Bhagat N, Mittal S, Ryaboy D (2014) Storm@twitter. In: SIGMOD

Vitter JS (1985) Random sampling with a reservoir. ACM Trans Math Softw 11(1):37–57

Xin R, Rosen J. Project Tungsten: bringing Spark closer to bare metal. http://tinyurl.com/mzw7hew

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI

Zamanian E, Binnig C, Salama A (2015) Locality-aware partitioning in parallel database systems. In: SIGMOD

Zeng K, Gao S, Gu J, Mozafari B, Zaniolo C (2014a) ABS: a system for scalable approximate queries with accuracy guarantees. In: SIGMOD

Zeng K, Gao S, Mozafari B, Zaniolo C (2014b) The analytical bootstrap: a new method for fast error estimation in approximate query processing. In: SIGMOD

Zeng K, Agarwal S, Dave A, Armbrust M, Stoica I (2015) G-OLA: generalized on-line aggregation for interactive analysis on big data. In: SIGMOD