Evolving Databases for New-Gen Big Data Applications

Ronald Barber Christian Garcia-Arellano* Ronen Grosman* Rene Mueller
Vijayshankar Raman Richard Sidle Matt Spilchen* Adam Storm*
Yuanyuan Tian Pınar Tözün Daniel Zilio* Matt Huras*
Guy Lohman C. Mohan Fatma Özcan Hamid Pirahesh

IBM Research – Almaden *IBM Analytics

ABSTRACT

The rising popularity of large-scale real-time analytics applications (real-time inventory/pricing, mobile apps that give you suggestions, fraud detection, risk analysis, etc.) emphasize the need for distributed data management systems that can handle fast transactions and analytics concurrently. Efficient processing of transactional and analytical requests, however, require different optimizations and architectural decisions in a system. This paper presents the Wildfire system, which targets Hybrid Transactional and Analytical Processing (HTAP). Wildfire leverages the Spark ecosystem to enable large-scale data processing with different types of complex analytical requests, and columnar data processing to enable fast transactions and analytics concurrently.

1. INTRODUCTION

Due to the way they evolved, relational DBMSs have always been strongest performing transactions that ensure the classical *ACID* properties. The early literature defined how to achieve strict serializability and isolation of concurrent transactions, and the Two-Phase Commit protocol for achieving consistent commits of distributed transactions. Indexes on any column, not just a primary key, facilitate accessing individual rows in the *point* queries typical of transactions. But traditional DBMSs also developed important technologies for more complex analytics queries, notably the declarative Structured Query Language (SQL) and robust optimization of it, and multi-node parallelism for speeding long-running queries. More recently, DBMSs have significantly accelerated analytics queries with a sophisticated exploitation of multi-threaded parallelism, compression, large main memories, and especially column stores [22, 26, 30, 33].

Nonetheless, DBMSs have their weak spots, too. A DBMS is a closed system that exclusively owns its data, which must be loaded into its proprietary format and slows retrieval for data-hungry applications such as Machine Learning. Replication of data is often provided as an afterthought for disaster recovery, with availability usually limited to those nodes having physical connectivity to the single copy and dependent upon the coordinator of Two-Phase Commit being up. Generally, the CAP Theorem states that strict consistency makes availability very hard [23]. Scale-out beyond a few hundred nodes of relational DBMSs is rare, and elasticity is limited because data is pre-allocated to, and stored on, a pre-determined (set of) nodes.

These weaknesses largely motivated the recent development of Big Data Platforms such as Hadoop [5] and now Spark [11], which were designed almost exclusively for performing complex and long-running analytics, such as Machine Learning, cost-effectively on extremely large and diverse data sets. These systems promote a much more open environment, both of functions and de facto standard data formats such as Parquet, allowing any function to readily access any data without having to go through a centralized gate-keeper. By routinely replicating data by default, usually asynchronously (e.g., with *eventual consistency* semantics), these systems built in high availability, scale-out, and elasticity from their inception.

However, Big Data platforms have their own shortcomings. Transactions (especially update-in-place) and point queries are largely ignored in Spark, thereby relegating ingest of data to simpler key-value stores such as Cassandra [4] and Aerospike [1]. Some of these stores may provide the high ingest rates required to capture data from new Internet of Things (IoT) applications, but to achieve this, have relaxed isolation levels and have embraced weaker *eventual consistency* of copies on independent nodes. They also index only a primary key, limiting point queries to those that specify that key. Lastly, they have limited or no SQL functionality, which is often added as almost an afterthought (e.g., Hive [34]), with weak query optimizers.

This paper argues that the Big Data world needs transactions. We present Wildfire, a design and initial prototype to bring ACID transactions, albeit with a weaker form of snapshot isolation, to the open analytics world of Spark. Wildfire exploits Spark for performing analytics by: (1) utilizing a non-proprietary storage format (Parquet), open to any reader, for all data; (2) using and extending Spark APIs and the Catalyst optimizer for SQL queries; and (3) automatically replicating data for high availability, scale-out performance, and elasticity (creating an AP system). Wildfire augments these Spark hallmarks with critical features from the traditional DBMS world, including: (1) ACID transactions with snapshot isolation, making the latest committed data immediately available to analytics queries; (2) the abil-

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

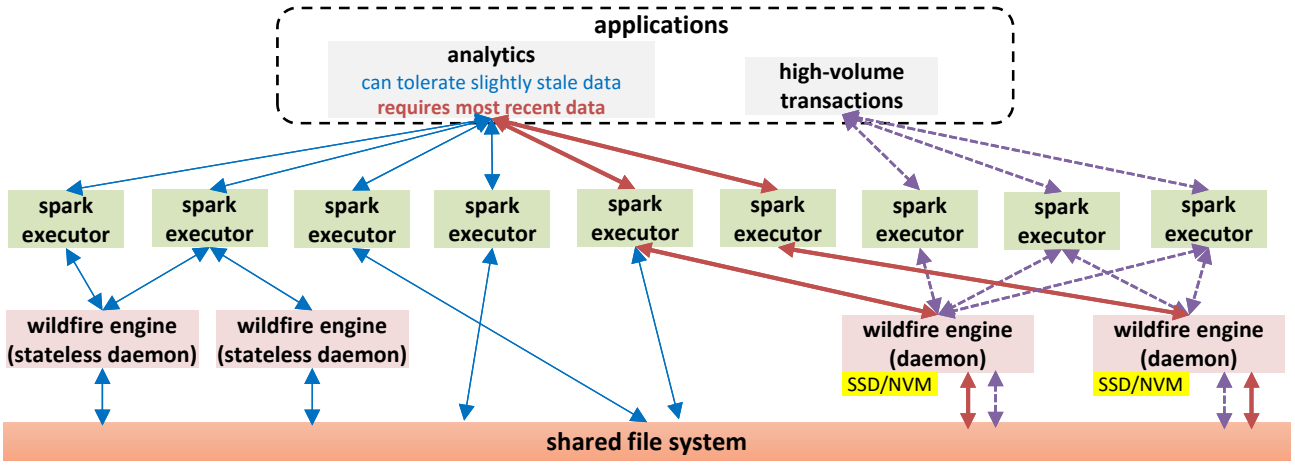

Figure 1: Wildfire Architecture.

ity to index any column for fast point queries; (3) exploiting recent advances for accelerating analytics queries by orders of magnitude, including compression *on the fly*, cache-aware processing, automatic creation and exploitation of synopses, (a form of) column-wise storage, and multi-threaded parallelism; and (4) enterprise-quality SQL, including more robust optimization and *time travel* that permits querying historical data *AS OF* a particular time.

2. WILDFIRE ARCHITECTURE

Figure 1 shows the Wildfire architecture, which has two major pieces: Spark and the Wildfire engine. Spark serves as the main entry point for the applications that Wildfire targets, and provides a scalable and integrated ecosystem for various types of analytics on big data, while the Wildfire engine accelerates the processing of application requests and enables analytics on newly-ingested data.

2.1 Processing of requests

All requests to Wildfire go through Spark APIs, except for a native OLTP API for the Wildfire engine, discussed later. Each request spawns Spark executors across a cluster of machines whose nodes depend upon the type of that request. The majority of the nodes in the cluster execute only analytical requests, and require only commodity server hardware (the solid arrows in Figure 1 show the request and data flow in these nodes). Other, beefier nodes, with faster local persistent storage (SSDs or, soon, NVRAM) and more cores for increased parallelism, handle concurrently both transactions and analytical queries on the recent data from those transactions (the dashed arrows in Figure 1 show the request and data flow in these nodes).

Wildfire's engine is based on columnar processing that is similar to DB2 with BLU Acceleration [33]. Each Wildfire engine instance *daemon* is connected to a Spark Executor. There are two types of engine daemons: stateful and stateless. The stateful daemons handle both transaction and analytics requests against the latest data. The stateless daemons, on the other hand, execute only analytics queries on the (much more voluminous) older data.

To speed ingest through parallelism, non-static tables in the system are sharded across nodes handling transactions based upon a prefix of a primary (single-column or composite) key. A table shard is also assigned to (a configurable number of) multiple nodes for higher availability. A stateful engine daemon on a node is responsible for the ingest, update, and lookup operations on the data assigned to that node, whereas the stateless engine daemons can read any data that is in the shared file system for analytical queries. A distributed coordination system (e.g., ZooKeeper[1]) manages the meta-information related to sharding and replication, and a catalog (e.g., HCatalog[2]) maintains the schema information for each table.

Wildfire openly allows any external reader to read data ingested by the Wildfire engine using Spark APIs without involving the Wildfire engine component, but that reader will be unable to see the latest transactional data stored on the stateful daemons. Further, to satisfy applications that need huge ingest rates, Wildfire provides a native API for the engine, where the insert requests to each table are kept as prepared statements after their initial invocation.

2.2 Processing and storage of data

Figure 2 illustrates the data life cycle in a shard replica in Wildfire. Each transaction in the Wildfire engine keeps its un-comitted changes in a transaction-local *side-log* composed of one or more log blocks. Each log block can contain transactions for only one table. At commit time, the transaction appends its side-log to the *log*, which is kept both in memory and persisted on disk (SSD or NVRAM). In addition, this side-log is copied to each of the other nodes that is responsible for maintaining a replica of that shard's data, for availability.

While any replica of a shard can process any transactional request for that shard, one of the replicas periodically invokes a *grooming* operation. This operation scans the log and groups together the log blocks from multiple (committed) transactions for the same table, creating larger groomed blocks containing data only from a single table. In addition to merging log blocks, grooming also performs some data cleansing that will be discussed in detail later. The groomed data blocks are then flushed to both the local SSD for fast reads and a distributed file system (e.g., HDFS [6], S3 [3], Swift [16], ...) so that other nodes can also access them. After a grooming pass, the groomer prunes the log records it has successfully groomed.

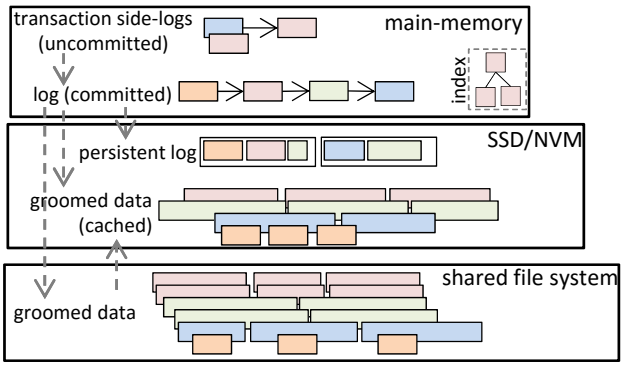

Figure 2: Data lifecycle in Wildfire.

The input source for queries in the Wildfire engine is both the (shared) groomed data and the (shard-local) log. In other words, each engine instance can read any groomed data regardless of its shard, but can only access log records for shards for which it is responsible. To avoid potential duplicates in this input stream while scanning both the log and groomed blocks, the engine checks the last groomed point in the log at the beginning of each query. The isolation level for queries who demand the latest data (dark red arrows in Figure 1) is snapshot isolation.

All tuples of a table are stored using the Parquet [9] format in both log and groomed blocks. Therefore, each block contains all column values for a set of rows of the table, and the values are stored in column-major format within the block, facilitating column-store-like access to just those pages containing columns referenced in a query, for larger, paginated blocks. The Parquet layout and local compression allow the data blocks to be fully self-contained.

3. TRANSACTIONS

Despite adopting columnar data procesing, the Wildfire engine is not just a query processor or accelerator for the Spark ecosystem. It is also designed to support transactions with inserts, updates, and deletes.

Wildfire targets high availability across multiple data centers, with tolerance for network partitioning. Therefore, it cannot give consistency semantics in which each read sees all prior writes [23]. Existing highly available systems such as Cassandra [4] normally provide either eventual consistency or forced multi-server quorum reads.

However, eventual consistency is painful for the application-writer. Consider two successive queries from an application. The first query may get results that are missed in the second query if it is routed to an alternate server that lags behind. Quorum reads, which perform redundant reads from multiple servers, are a reasonable alternative. However, they are not only infeasible for OLAP-style transactions that may read thousands, millions, or billions of records, but also costly for single-key fetch queries.

Wildfire targets both high availability and ACID, which is infeasible. Therefore, Wildfire adopts last-writer-wins (LWW) semantics for concurrent updates to the same key and snapshot isolation of quorum-readable content for queries, without having to read the data from a quorum of replicas to satisfy a query. The remainder of this section describes some of the design choices and methods to reach this goal.

3.1 Writes: Inserts, Updates, and Deletes

It is impractical to send changes directly to the shared file system, which is typically append-only and optimized for large blocks. Therefore, as Section 2 describes, Wildfire first writes (and persists at commit) the transactional changes to local storage. Only a background *grooming* process propagates them to the shared file system, in a batched fashion.

The logs for a table in Wildfire are sharded across processing nodes using a key composed of one or more columns of the table. In addition, for high availability, these shard logs are replicated to multiple nodes (a minimum of 3). The writes (inserts, updates, deletes) of a transaction are sent to any node that contains a shard replica. At commit, the changes for the transaction are applied to the local logs and then replicated.

3.1.1 Replication

In the case of synchronous replication (at least to a quorum) Wildfire faces the danger of losing availability. Asynchronous replication, on the other hand, might suffer from inconsistency – *e.g.*, a query that immediately follows a transaction may not see that transaction's writes if it is routed to a different node than the transaction.

In Wildfire, every (write) transaction performs a status-check query at the end: one that simply waits until the writes of that transaction to be replicated to a quorum of nodes. Similarly, the read-only queries return quorum-replicated data.

At a poorly connected node, the status-check may time out. To sustain high-availability in this case, Wildfire returns to the client with a *pending* message, indicating that the transactions position (in the serializable order of transactions) is unknown until a future point in time when the status-check succeeds. This behavior mimics the best practice in the financial industry, where the ATM transactions are allowed to proceed during network disconnection, with a disclaimer that the order of transactions are going to be resolved subsequently.

This *delayed-commit* semantics does come at a high cost: *one cannot check integrity constraints at commit.* Hence, concurrent updates to the same key based on prior values are going to suffer from the lost-update problem. Wildfire resolves this by adopting the LWW semantics as mentioned above.

In the case where a client receives a time-out message, Wildfire offers a *SyncWrite* option. If the client confirms that their writes are idempotent, Wildfire automatically reissues any timed-out writes on other nodes, until they succeed. The kind of applications that require AP from CAP, tend to have writes that are idempotent. If a non-idempotent write times out or the client connection breaks, the client is left hanging, as there is no easy way to figure out whether that write has succeeded.

3.1.2 Shards

Each table must have a primary key that is made up of a subset of the columns of the sharding key. This is sligtly different than the constraint of having a prefix of the primary key as the sharding key like systems like Megastore [20] and Cassandra [4] adopt. Inserts of pre-existing keys are treated as updates, and deletes are treated as inserts of tombstones. Any update, delete, or insert results in just an insert of a

new version, with a begin and end timestamp (*beginTS* and *endTS*. The beginTS is just the commit timestamp, and the endTS is the beginTS of the next version of that key.

Wildfire's client-side logic accepts and partitions bulk insert requests based on the sharding key to determine the target shard(s). These partitioned inserts are sent to a replica for each shard with some affinity, but with the ability to automatically fail-over to another replica to handle error scenarios. The partitioned inserts are cached by the client library until a SyncWrite is requested and successful. Should a failure occur in this phase, the client library will re-submit the cached partitioned inserts. Should memory pressure occur at the client, the library itself will trigger a *SyncWrite* request.

3.1.3 Conflict Resolution

Each shard has a designated groomer that runs at one of its replica nodes. The groomer merges, in time order, the logs from each replica of the shard and creates Parquet-format files in the shared file system for the data modifications. Indexes on the primary keys are built during grooming to detect multiple versions of the same data at a later phase called *post-grooming*. This periodic post-grooming operation performs conflict resolution where it sets the endTS of the previous version to the beginTS of next version for records with the same primary key. This post-grooming operation also replaces the files in the shared file system as needed. Queries which find unresolved duplicates would then know to perform special handing by looking up these keys to determine the correct version to use, thus implementing LWW semantics.

Each instance of Wildfire tracks the log replication points for all replicas and computes a current high-water mark of the data that is quorum-visible. Queries are then able to achieve quorum-consistent reads without accessing the same data at multiple replicas.

The beginTS is a local wall-clock time of the commit: but changes from different nodes can replicate at arbitrary speeds. So changes are ordered within each grooming cycle by a commit timestamp, but we use the groom cycle time as a high-order timestamp for the set of groomed changes, thus eliminating any need to re-order late replicated changes back into the already groomed ordering of history. This, in a sense, pushes the effective commit time to the quorum readable time.

3.2 Reads

The log (local or replicated) has only committed transactional changes. However, queries (including the grooming query) need to see all quorum-written changes. So we use a high-water mark of quorum-visible portions of the replicated logs. Depending on the currency of data needed by queries, groomed data may be all that is needed. However, certain classes of queries read the log entry changes along with the groomed data. The grooming process itself reads only the log entry changes to perform its processing.

Snapshot isolation needs a system-generated predicate: beginTS $\leq$ *snapshotTS* $<$ endTS. The snapshotTS is usually the transaction start time, but can be changed to allow time-travel. The begin timestamp, as stated earlier, is set when the record is committed and then updated again at grooming time to pre-pend the groom timestamp. The end timestamp is initialized to infinity, except in the case of deletes, and left unchanged at groom time unless more than one occurrence of the primary key occurs in the grooming cycle, in which case the earlier entries will have their end timestamps set to the begin timestamps of their replacements.

This does not address changes to the end timestamp due to updates of older rows that had already been groomed earlier. Those are addressed in two parts. First, the periodic post-grooming process will rewrite blocks, filling in the end timestamp based on key. To handle changes in tail blocks, Wildfire maintains a hash table tracking key versions (beginTS and rowID). Queries probe this hash table if the end timestamp is infinity for a record.

4. ANALYTICS

Apache Spark provides an extensive ecosystem for big data analytics, streaming, machine learning, and graph processing. We integrate Wildfire into the Spark environment in order to build on top of its existing capabilities. Wildfire enhances Spark with the missing support for OLTP and improves its OLAP performance.

In this section, we describe the major extensions of Wildfire to Spark: (1) the new OLTP interface `OLTPContext`, (2) extensions to the Spark Catalyst optimizer and the existing OLAP `SQLContext` to enable the push-down of queries into the Wildfire engine, and (3) our support of user-defined function (UDF) and user-defined aggregate functions (UDAF) in Wildfire.

4.1 New Interface for OLTP

In order to provide HTAP functionality, we need support for OLTP operations, i.e., point queries and inserts or upserts. However, this functionality is currently missing in the Spark ecosystem. Wildfire builds a new OLTP interface that can be used by Spark applications, called `OLTPContext`. For now, this interface is kept separate from Spark's existing OLAP interface, `SQLContext`. The two interfaces may be unified in future versions of Spark. Our OLTP API plays very well with the different components of Spark. For example, we can use it together with Spark Streaming for high-rate inserts from streaming data sources. We can also use this OLTP API together with Spark SQL for HTAP.

The `OLTPContext` accesses and caches the coordination service to retrieve the configuration state of the backend Wildfire cluster, i.e., the set of Wildfire engines and the shards they host, as well as a reference to the catalog service. In order to route a transaction to the right shard, the OLTP needs to uniquely identify the shard, e.g., through the sharding key for an insert or a statically evaluable predicate in a point query. Our initial prototype does not yet support transactions that span multiple shards, but plan in the future. Once the shard is determined, the `OLTPContext` routes the reads and writes to the appropriate Wildfire engines that host the corresponding shared or shard replicas. The context obtains the current shard-to-node assignment from the coordination service. If the `OLTPContext` is unable to identify the shard from the query, e.g., the point-query does not have a predicate on the sharding key, or does not identify a unique shard, or cannot be statically evaluated, the `OLTPContext` broadcasts point queries to all Wildfire engines. This, however, comes at a higher cost and breaks transactional isolation in our current prototype, as it may result in a cross-shard transaction.

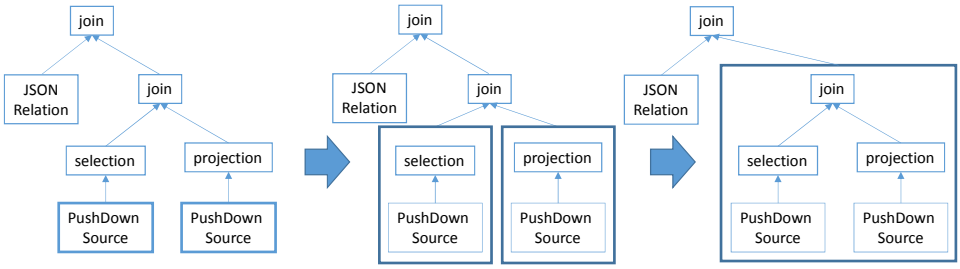

Figure 3: Bottom-up build up of the pushdown plan

We also handle node failures in the `OLTPContext`. For example, if a node that is responsible for the shard of a number of rows that are being inserted fails, we try to re-insert those rows to one of the replica nodes and update the host-to-shard mapping.

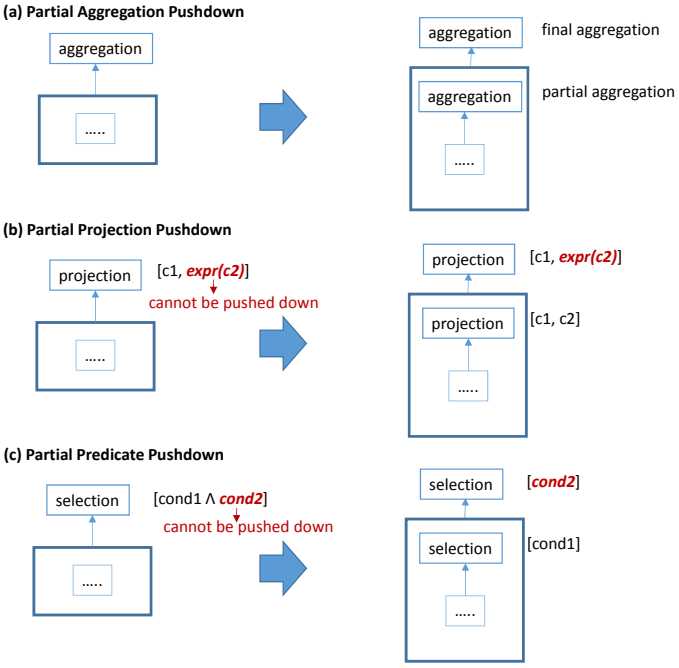

Figure 4: Examples of partial pushdown

4.2 Extensions to Spark SQL for OLAP

For OLAP, we want users to be able to query Wildfire tables using the same Spark SQL interfaces (either via Spark DataFrames or SQL) as they do for regular Spark tables. Moreover, we want to be able to use both Wildfire tables and normal Spark tables in the same query, e.g., joining a Wildfire table with a JSON table.

We achieve this seamless integration by extending both Spark SQL's Data Sources API and the Catalyst query optimizer. The Data Sources API provides a way to access data sources outside Spark through Spark SQL in an easy and pluggable manner. Spark's Catalyst optimizer currently is able to push down projection and filtering operations to the data sources, if supported by the sources, through the Data Sources API. However, our Wildfire engines provide more advanced query capabilities for Spark SQL to leverage. We can push down even more complex operations such as joins and partial aggregations, as well as user-defined functions and aggregates.

These extensions to the Data Sources API and the Catalyst optimizer are general and not just for Wildfire. Arbitrarily complex queries can be pushed down to any data source that implements our API extensions, as this approach allows the source to decide what plans can be pushed down. With this general pushdown approach to a remote source, we essentially enable Spark to be a federation engine for big data systems.

4.2.1 *Extension to Data Sources API.*

To allow more complex pushdown, we introduce a new type of data source, called `PushDownSource`, to the Data Sources API. The API provided by `PushDownSource` allows a data source to express its pushdown ability to the Catalyst optimizer. Given a Spark logical plan (a tree-structured logical query plan), a data source can express, through this API, whether the entire logical plan can be executed in the source or not. If a plan cannot be executed in the source, this API further provides a way to examine whether individual expressions inside a plan can be supported by the source, which is important to allow partial push-downs (details will be provided below).

4.2.2 *Extension to Catalyst Optimizer.*

We also extend Spark's Catalyst optimizer to enable the pushdown analysis for a data source that implements the `PushDownSource` API. More specifically, we add a number of rewrite rules to the logical optimization phase of the query optimization. Each rule rewrites a query plan to a logically equivalent plan, in the usual way. Together, they identify and build up the pushdown plan in a bottom up fashion, as shown in Figure 3. We start with leaf nodes that are `PushDownSource`. They represent the base tables in the target data source. Obviously, they can be pushed down to the source. Then we look at the parent of each `PushDownSource`. By using the extended API, Catalyst can know whether the subquery represented by the parent can be pushed down to the source or not. If so, we construct a new leaf node to replace the parent, and track the pushdown plan inside the leaf node. In case of a join, we push down the join only if both children are pushed down already, and the join itself can be pushed down (e.g., colocated joins). This process is continued until a fixed point is reached (no change to the logical plan occurs).

In a number of cases, we cannot push down the entire subquery represented by a tree node, but we can rewrite the plan so that part of the subquery can be pushed down.

Figure 4 illustrates three most common examples of partial pushdown.

Partial Aggregation Pushdown: As many data sources, including Wildfire, do not have the ability to transfer data among themselves for query processing, aggregate functions cannot be fully pushed down. In this case, we rewrite an aggregation plan into a partial aggregation followed by a global aggregation, and push down the partial aggregation. For example, to support `count(.)` for Wildfire, it is rewritten into a partial `count(.)` that is executed on all the Wildfire engines, followed by a global `sum(.)` that is carried out in Spark.

Partial Projection Pushdown: For projection, if the list of column expressions contains one or more expressions not pushdown-able, we split the projection plan into two consecutive projections. The first is pushed down to the source with the basic columns needed for all the expressions, and the second is executed in Spark for evaluating the actual expressions.

Partial Predicate Pushdown: If a conjunctive predicate contains one or more sub-predicates that cannot be pushed down, we only push down the pushable sub-predicates, and form a new selection node with the non-pushable sub-predicates.

4.3 Using OLTPContext and SQLContext for HTAP

Applications that require HTAP instantiate both the new `OLTPContext` and the `SQLContext` in the Spark driver. This allows them to submit analytics queries through our extended `SQLContext`, and point queries as well as inserts via the `OLTPContext` to Wildfire. An OLAP query is assigned a snapshot that is based on the required maximum tolerable staleness of the data. If that staleness is shorter than the grooming interval (typically just a second or two, but this is configurable), the query is either delayed until grooming has caught up to the snapshot, or the query must be sent to the Wildfire engine nodes to be processed from the logs on the node-local SSDs. Unless shard (partition) elimination can be applied, the query must be sent to all Wildfire engine nodes. Therefore, analytic queries with such short staleness requirements are more expensive and may negatively affect the transaction throughput of pure OLTP queries. This, however, is no different from traditional database systems for which admittance control is used to strike a balance between the resource usage of analytical and transactional queries. OLAP queries that can tolerate a staleness that is longer than Wildfire's grooming interval can be processed more inexpensively with data read from the shared file system by any nodes.

4.4 User-defined Functions and Aggregates

A key feature in Spark and Spark SQL is the extensibility from the end-user's perspective. User-defined scalar functions (UDFs) and user-defined aggregate function (UDAF) can be defined and used in queries. The use of anonymous functions (lambdas) in Java 8 and Scala makes this extremely powerful while being easy to use. It is therefore crucial for Wildfire to also support UDFs and UDAFs and to be able to execute them inside the engine. Scalar UDFs can be used in the *select* and the *where* clause of queries. When used inside aggregate functions and predicates, or are user-defined aggregation function themselves, they can reduce the amount of data returned to Spark. UDFs can contain logic that is hard to express in SQL (e.g., decision trees, machine learning models used for scoring, or even deep learning models). Wildfire supports UDFs and UDAF in Java bytecode from Java and Scala and executes them in embedded Java virtual machines that run inside the Wildfire engines. Since the Wildfire engines are implemented in a native code environment, it will be easier to add hardware accelerators such as GPUs and FPGAs in order to run UDFs with even more complex models.

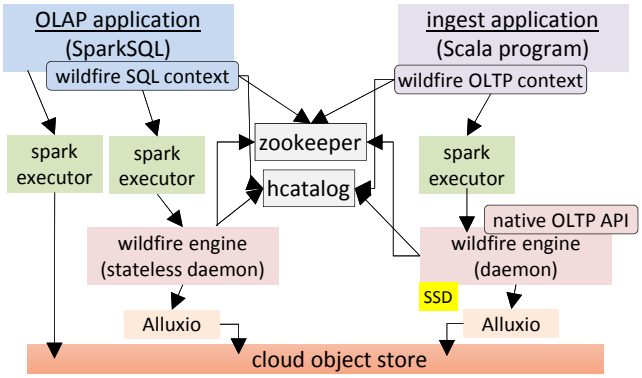

Figure 5: Current Wildfire prototype.

5. PROTOTYPE

We presented the initial prototype of Wildfire in SIGMOD 2016 [21]. Since then we enhanced this prototype toward our end goal (depicted in Figure 1). Figure 5 shows the current state of Wildfire.

SparkSQL is the entry point for analytical applications, and a Scala-based interface is used for OLTP applications (currently just ingest requests). As mentioned in Section 2.1, Wildfire also provides a native API for the engine, which was used during the SIGMOD demo for ingest requests as our scala API for OLTP was primitive then. ZooKeeper is used as the coordination service and HCatalog is the primary source for catalog information. The engine and client layer contact ZooKeeper for sharding information. The engine also contacts ZooKeeper to learn about the state of replicas and the last groom points for each shard. The fast local storage for the engine, where heavy ingest requests are handled concurrently with analytical requests, is SSDs. Grooming writes the data blocks both to SSDs and the shared file system. The blocks in SSDs are evicted based on an LRU policy (groom time) and the space budget of the SSDs. The shared distributed storage system used in the prototype is an object store with Alluxio [2] serving as a cache on top.

We are currently working on exposing the OLTP interface of the Wildfire engine to Spark, so that applications running inside Spark can have access to the full HTAP functionality. In addition, we are extending the Wildfire engine to support more complex data types (e.g., JSON, arrays). Lastly, we are improving the indexes in Wildfire to support fast point queries on both primary and secondary indexes, and working on enabling more complex transactions.

6. RELATED WORK

Over the last decade, although several SQL processing systems have been developed, especially in open-source [18], none process both analytical as well as transactional workloads. Most of these systems, including Hive [34], Impala [29], HAWQ [25], Big SQL [27], and Spark SQL [19], have all focused on analytics over HDFS data initially. Since HDFS and Hadoop's focus was batch processing, data was also ingested in batches. For applications that required updates and faster insertion rates, noSQL systems provided an alternative. HBase [7, 35] and Cassandra [12, 4] are two of the most popular noSQL systems for this purpose. However, this led to lambda architectures where transactional systems were separate from analytical systems. The purpose of Wildfire is to provide a single unified platform for both transactional and analytical processing.

Over the years, some of these initial systems, like Hive and Impala, also included support for updates. As of very recently, Hive supports ACID transactions [13], but with several limitations, such as not supporting explicit transaction begin, commit, and rollback statements. The integration of Impala [29] with the storage manager Kudu [8], on the other hand, allows the SQL-on-Hadoop engine to handle updates and deletes reducing the pitfalls of using HDFS and HBase for transactions and analytics, respectively. HAWQ [25] supports snapshot isolation, as it uses PostgreSQL as its underlying processing engine. It only allows appends, and transactions can only commit on the master node, a central fixed node. Hence, these systems are not meant to support a high volume of transactions but rather batch inserts and slowly changing dimensions that are typical in classical data warehouse workloads.

There are other systems, like Splice Machine [17] and Phoenix [10] that allow updates and transactions. These systems provide SQL processing for data stored in HBase tables, and as a result rely on HBase for the updates. Splice Machine even supports ACID transactions. However, these systems do not provide fast OLAP capabilities because the scans over HBase tables are quite slow. Most often, the data is transformed into a more analytical-friendly format, such as Parquet, and processed by one of the other SQL engines, such as Hive, Impala, or SparkSQL. This data copying is both error-prone and costly, and also it does not allow analytics to work on the latest data.

Oracle [31], SAP HANA [26], and MemSQL [14] are among the systems that support hybrid analytical and transactional workloads as stand-alone engines, but they use different formats for data ingestion and analytics. As a result, the latest committed data is not available to analytical queries right away, or else accessing the latest data requires a costly join between row-store and column-store tables. In Wildfire, by using a single data format for both data ingestion as well as analytics, we enable analysis on the latest committed data right away. HyPer [28] also supports hybrid workloads using multi-version concurrency control, and exploiting machine code generation with LLVM for very optimized single-threaded performance. However, it is not clear how HyPer behaves in a large-scale distributed setting.

The data lifecycle of Wildfire going from memory to SSD/NVM and to a shared file system is inspired by the design for data movements and compactions in systems like BigTable [24] and MyRocks [15]. However, Wildfire is not based on LSM-trees [32].

7. CONCLUSIONS

We presented the Wildfire system, which is designed to handle high-volume transactions while executing complex analytics queries concurrently in a large-scale distributed big data platform. The analytical queries are issued via the Spark SQL API, and a Spark Executor is connected to Wildfire's columnar engine on each node. The connection to Spark exposes the analytics capabilities of Wildfire to the entire Spark ecosystem, including graph processing and machine learning. Wildfire also extends Spark's Catalyst optimizer to perform complex push-down analysis, and generates compensation plans for the remaining portions of the analytics queries that cannot be pushed down into Wildfire's columnar engine.

8. REFERENCES

[1] Aerospike. http://www.aerospike.com/.
[2] Alluxio. http://www.alluxio.org/.
[3] Amazon S3. https://aws.amazon.com/s3/.
[4] Apache Cassandra. http://cassandra.apache.org.
[5] Apache Hadoop. http://hadoop.apache.org/.
[6] Apache Hadoop HDFS. http://hortonworks.com/apache/hdfs/.
[7] Apache HBase. https://hbase.apache.org/.
[8] Apache Kudu. https://kudu.apache.org/.
[9] Apache Parquet. https://parquet.apache.org/.
[10] Apache Phoenix. http://phoenix.apache.org/.
[11] Apache Spark. http://spark.apache.org/.
[12] DataStax Spark Cassandra Connector. https://github.com/datastax/spark-cassandra-connector.
[13] Hive Transactions. https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions.
[14] MemSQL. http://www.memsql.com/.
[15] MyRocks. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/.
[16] OpenStack Swift. https://www.swiftstack.com/product/openstack-swift.
[17] Splice Machine. http://www.splicemachine.com/.
[18] D. Abadi, S. Babu, F. Özcan, and I. Pandis. Tutorial: SQL-on-Hadoop Systems. *PVLDB*, 8:2050–2051, 2015.
[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.
[20] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.
[21] R. Barber, M. Huras, G. M. Lohman, C. Mohan, R. Mueller, F. Özcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, A. Storm, Y. Tian, and P. Tözün. Wildfire: Concurrent Blazing Data Ingest and Analytics. In *SIGMOD*, pages 2077–2080, 2016.
[22] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[23] E. A. Brewer. Towards Robust Distributed Systems. In *PODC*, pages 7–, 2000.

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[25] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop. In *SIGMOD*, pages 1223–1234, 2014.

[26] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE DEBull*, 35(1):28–33, 2012.

[27] S. Gray, F. Özcan, H. Pereyra, B. van der Linden, and A. Zubiri. IBM Big SQL 3.0: SQL-on-Hadoop without compromise. https://public.dhe.ibm.com/common/ssi/ecm/sw/en/sww14019usen/SWW14019USEN.PDF, 2015.

[28] A. Kemper and T. Neumann. HyPer – A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011.

[29] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[30] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

[31] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, T. Lahiri, J. Loaiza, N. Macnaughton, V. Marwah, A. Mullick, A. Witkowski, J. Yan, and M. Zait. Distributed Architecture of Oracle Database In-memory. *PVLDB*, 8(12):1630–1641, 2015.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[33] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 6:1080–1091, 2013.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, pages 996–1005, 2010.

[35] Z. Zhang. Spark-on-HBase: Dataframe Based HBase Connector. http://hortonworks.com/blog/spark-hbase-dataframe-based-hbase-connector.