Wildfire: Concurrent Blazing Data Ingest and Analytics

Ronald Barber Matt Huras* Guy Lohman C. Mohan Rene Mueller
Fatma Özcan Hamid Pirahesh Vijayshankar Raman Richard Sidle
Oleg Sidorkin• Adam Storm* Yuanyuan Tian Pınar Tözün

IBM Research – Almaden *IBM Analytics •IBM DemandTec

ABSTRACT

We demonstrate Hybrid Transactional and Analytics Processing (HTAP) on the Spark platform by the Wildfire prototype, which can ingest up to ≈6 million inserts per second per node and simultaneously perform complex SQL analytics queries. Here, a simplified mobile application uses Wildfire to recommend advertising to mobile customers based upon their distance from stores and their interest in products sold by these stores, while continuously graphing analytics results as those customers move and respond to the ads with purchases.

1. INTRODUCTION

Traditionally, OLTP (On-Line Transactional Processing) systems have been separated from OLAP (On-Line Analytics Processing) systems because of the incompatibilty in their requirements: resource-hungry, long-running analytics queries interfere with the latency requirements of short-running transactions. However, recent advances in speeding up analytics queries [6, 7, 8, 9] now make it possible to combine these systems into one integrated system called Hybrid Transactional and Analytics Processing (HTAP). Doing so permits analytics to see the latest transactions, reducing the business risk of making wrong decisions, such as fraud detection, on stale data. It also dramatically increases business agility, permitting reports and analytics on operational data to perform real-time pricing, inventory, and supply-chain management. Thirdly, combining systems simplifies the system landscape and reduces operational costs.

Concomitantly, the increased availability of “Big Data” sources, tools, and processing power has spawned new applications that seek to understand and exploit individual customer behavior through analysis of that data. This analysis may include any combination of machine learning, graph processing, and stream processing, as well as more traditional forms of summarization, analytics, and reporting. Data volumes are increasingly driven by the wide deployment of inexpensive sensors, especially on mobile, geo-located platforms, which now generate large volumes of data at much higher rates than traditional transactional data, in aggregate exceeding millions of rows per second. These requirements have largely driven these new applications toward the Spark platform [10, 2], which offers: massively parallel execution on clusters of inexpensive commodity machines; in-memory processing that is 10x to 100x faster than Hadoop; flexible programming interfaces; and rich library support for machine learning, graph processing, and data streaming.

This demonstration presents Wildfire, which seeks to support this new class of scalable HTAP applications that require ultra-high data ingest rates in excess of 1 million inserts per second per node while concurrently performing analytics queries at speeds comparable to the latest Relational DBMS (RDBMS) engines [6, 7, 8, 9], about one half order of magnitude faster than existing Spark SQL [4]. We leverage the Spark infrastructure to provide a massively parallel and elastic HTAP solution that enables different types of analytics via the Spark ecosystem (Spark SQL, MLLib, GraphX, etc), but can also ingest data at very high rates. Our longer-term goal is to provide ACID transactions at these rates, and even to run analytics queries as part of such transactions (true HTAP).

To demonstrate Wildfire, we chose a hypothetical but plausible system that offers promotions to mobile customers whose new locations are continuously ingested. The promotions are offered as advertisements that are decided by a simulated machine learning model that is a function of the proximity of each user to store locations selling certain products, the remaining advertising budget for each store, the time since the last promotion displayed to that user, etc. The demo shows the rates at which user locations are ingested, while showing the results of several concurrently-executing analytic queries that summarize the data either geographically, by store, or by product category.

Our contributions include: (1) ingesting data to a distributed, persistent store (Solid-State Drives on commodity servers) at rates around 6 million / second / node; (2) concurrently performing multiple complex SQL analytic queries involving joins, grouping, and aggregation over this distributed data, using Spark SQL, as soon as it is persisted; and (3) simulating the exploitation of machine learning models derived from this data to make real-time business decisions based upon real-time location and budgetary information.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2899406

2. WILDFIRE ARCHITECTURE

Figure 1 shows the current Wildfire architecture. We have a cluster of *Wildfire engines* running concurrently to

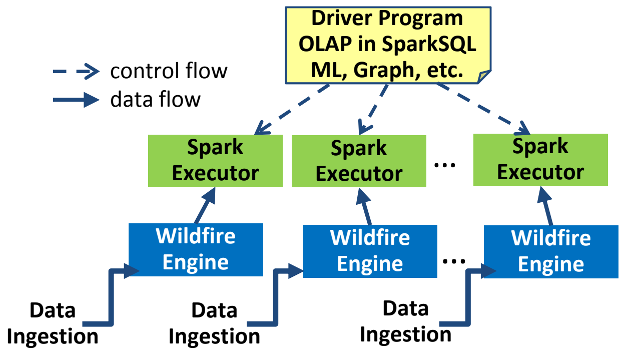


Figure 1: Wildfire Architecture for Demo.

ingest data and serve OLAP queries. Each table in our system is partitioned across the Wildfire engines, and each Wildfire engine ingests its partition of the data through a fast native API (marked as *Data Ingestion* in Figure 1). To enable distributed and complex analytics over large volumes of data, we integrate with Spark to run multiple Wildfire engines in a cluster, answering analytics queries through the Spark SQL interface. The Wildfire application *driver* runs in Spark and spawns *Spark executors* to coordinate the requests/results to/from the distributed Wildfire engines.

2.1 Wildfire Engine: Storage and Processing

Tables in Wildfire are stored in large blocks (up to 64 MB each) using a PAX layout [3] that is similar to Parquet [1]. As such, each block contains all column values for a given set of rows of the table, and the values are stored in column-major format within the block. Wildfire employs a block-local minus-coding scheme to compress the values in the block. The encoding is fixed in size (from 0 to 64 bits) within each block, and encoded values are written in fixed-sized chunks, with 512 values per chunk. The combination of the fixed-size encoding and chunking enables fast column scans on the encoded data. Given the PAX layout and local compression, the storage blocks are fully self-contained and portable. Therefore, these blocks are amenable to processing in a distributed compute environment such as Spark and storage within a cluster filesystem such as HDFS. In addition, one can efficiently append to an existing block and perform point-access to a data item in a block.

Wildfire uses a columnar query engine that has many similarities to the query engine of DB2 with BLU Acceleration [9]. Wildfire processes composite queries, such as joins, in multiple stages. Each stage scans a single input table. A query stage can have side-effects, for example by generating a hash table for the "build" of a hash join (to be probed during a subsequent stage that scans the outer table) or by inserting into another table. A query stage is executed as a sequence of operators, called *evaluators*. Evaluators employ vectorized processing, with each evaluator taking as input a set of vectors of values and producing one or more result vectors. Each query thread has its own evaluator sequence. Query threads perform work-stealing on the single input and fully process a batch of input rows through all evaluators before obtaining another batch.

To achieve high ingest rates, Wildfire provides a native request interface in the form of pre-compiled stored procedures for inserts to each table. In addition, the engine combines multiple insert requests and applies them in batched fashion using the vectorized processing described above. Blocks are appended to files when full or at the end of a batch insert.

Wildfire also uses non-partitioned hash joins and Concise Hash Tables, as described in [5]. In addition to column scans, hash joins, and inserts, Wildfire has support for many other evaluators, such as hash-based group by, predicate evaluation, expression evaluation, and updates of in-memory (non-persistent) indexes.

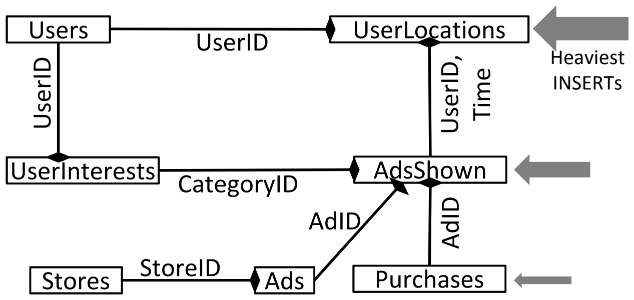


Figure 2: Wildfire Demo Application Schema.

2.2 Integration with Spark

To support distributed OLAP, we integrate the Wildfire engine into the Spark environment, as shown in Figure 1. We achieve this integration by implementing the Spark Data Sources API for Wildfire. Users issue their OLAP queries against the tables partitioned across the Wildfire engines through the Spark SQL interface, which submits the same query to all the Wildfire engines for processing and computes the final result. We configure the Spark driver to spawn one Spark executor for each Wildfire engine. To take advantage of locality, we also provide location hints to Spark so that the Spark executors are spawned on the same machines as the Wildfire engines. A submitted query is passed down from each Spark executor to its Wildfire counterpart through a TCP connection, and the Spark executor subsequently receives results back from the engine over the same connection. The final query result can then be further used for machine learning, graph, or other analyses supported by the Spark ecosystem.

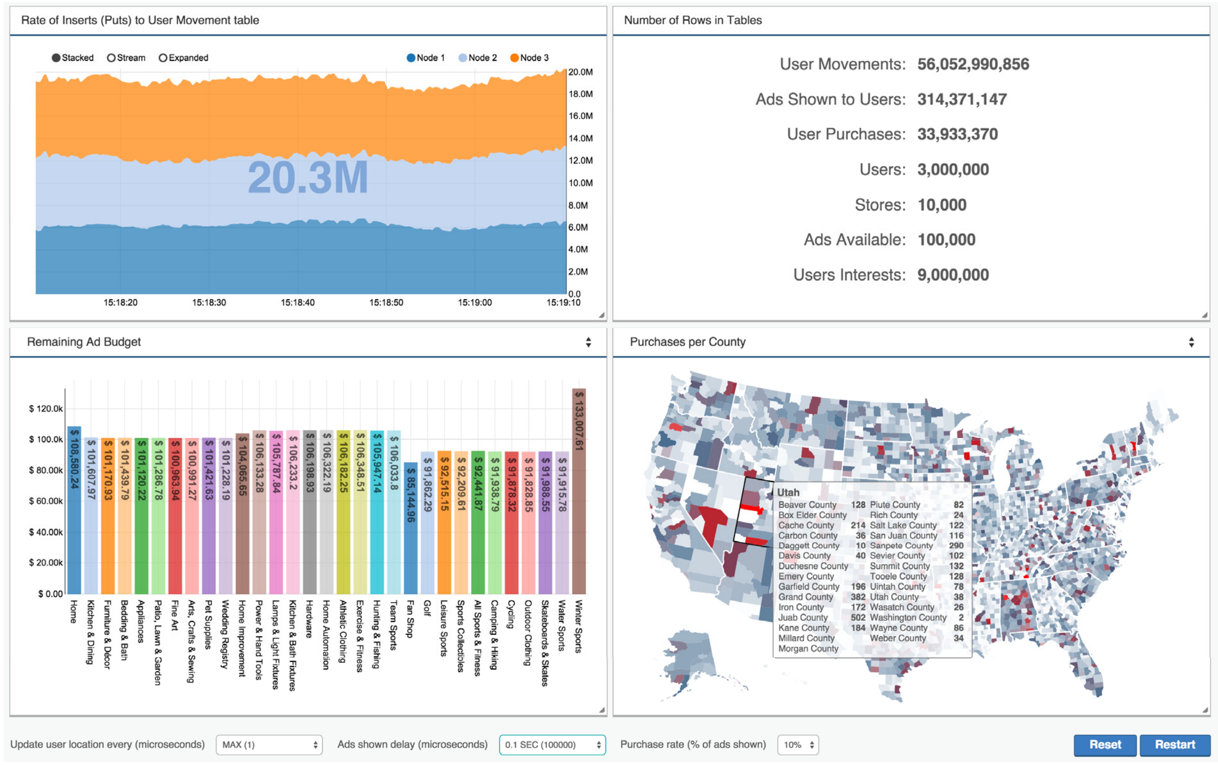
3. DEMO

3.1 Application

To drive the demonstration of Wildfire, we have implemented a simplified application to offer promotions (advertisements) for products to mobile users whose locations are ingested at very high rates and stored persistently in solid-state drives (SSDs) by Wildfire. Which ad is shown to each user, and which products each user purchases, are also stored in separate tables.

3.1.1 Schema

Figure 2 provides the schema diagram for the application; each rectangle corresponds to a table, and edges represent joins between tables, labeled with the join column(s). Figure 2 also highlights those tables with the highest ingest rates. A fixed number (3 million) of `Users` move randomly within a region, providing new `UserLocations` at sub-second



Number of Rows in Tables

User Movements:	56,052,990,856
Ads Shown to Users:	314,371,147
User Purchases:	33,933,370
Users:	3,000,000
Stores:	10,000
Ads Available:	100,000
Users Interests:	9,000,000

Home	$ 108,580.24
Kitchen & Dining	$ 101,607.97
Furniture & Décor	$ 101,170.93
Bedding & Bath	$ 101,439.79
Appliances	$ 101,120.22
Patio, Lawn & Garden	$ 101,286.78
Fine Art	$ 100,963.94
Arts, Crafts & Sewing	$ 100,991.27
Pet Supplies	$ 101,421.63
Wedding Registry	$ 101,228.19
Home Improvement	$ 104,065.65
Power & Hand Tools	$ 106,133.28
Lamps & Light Fixtures	$ 105,787.84
Kitchen & Bath Fixtures	$ 106,233.2
Hardware	$ 106,198.93
Home Automation	$ 106,322.19
Athletic Clothing	$ 106,182.25
Exercise & Fitness	$ 106,348.51
Hunting & Fishing	$ 105,947.14
Team Sports	$ 106,033.8
Fan Shop	$ 85,144.96
Golf	$ 91,862.29
Leisure Sports	$ 92,515.15
Sports Collectibles	$ 92,209.61
All Sports & Fitness	$ 92,441.87
Camping & Hiking	$ 91,938.79
Cycling	$ 91,878.32
Outdoor Clothing	$ 91,828.85
Skateboards & Skates	$ 91,988.55
Water Sports	$ 91,915.78
Winter Sports	$ 133,007.61

Utah

Beaver County	128	Piute County	82
Box Elder County		Rich County	24
Cache County	214	Salt Lake County	122
Carbon County	36	San Juan County	116
Daggett County	10	Sanpete County	290
Davis County	40	Sevier County	102
Duchesne County		Summit County	132
Emery County		Tooele County	128
Garfield County	196	Uintah County	78
Grand County	382	Utah County	38
Iron County	172	Wasatch County	26
Juab County	502	Washington County	2
Kane County	184	Wayne County	86
Millard County		Weber County	34
Morgan County			

Update user location every (microseconds) MAX (1)
Ads shown delay (microseconds) 0.1 SEC (100000)
Purchase rate (% of ads shown) 10%

Figure 3: Wildfire Demo Live Screen.

intervals determined by a system parameter. An in-memory hash-index keeps the latest UserLocation for a User. A fixed number (10,000) of Stores at fixed locations in the region sell a different small subset of 100,000 products, each of which is advertised by an Ad and each of which fits into one of 31 broad categories. Each User has three UserInterests in buying products from these categories (so UserInterests table contains 9 M rows); these UserInterests and the strength of interest are different for each User and are pre-populated. Other system parameters, which the demo viewer may modify, control the rates at which Ads are shown to Users (5 secs. to 1 microsec.) and at what rate Users make Purchases (10% to 70% of AdsShown). Users are randomly partitioned, whereas Stores and Ads are replicated at each partition. All other tables are co-partitioned with the Users, based on the *UserID*.

3.1.2 *Data Ingest (OLTP)*

Each new UserLocation for a User is randomly generated as an azimuth and distance from the last UserLocation, without following a road network. These new UserLocations drive the highest ingest rates, in excess of 20 M rows per sec. at the most frequent update rate (1 microsec.).

As these UserLocations are ingested, a complex analytics query, ShowAds, runs continuously to determine which Ads to show to each User next. The results of ShowAds, described in Section 3.1.3, are inserted into the AdsShown table. To avoid bombardment of users with promotions, not every new UserLocation results in an AdsShown insert, so its ingest rate is significantly less than that of UserLocation.

Once an Ad is shown to a User, the User "clicks through" to purchase the product with some probability set by the system parameter, causing an insert into the Purchases table. That parameter therefore determines the relative size of Purchases to AdsShown.

3.1.3 *Complex Analytics (OLAP) Queries*

Concurrent with these high ingest rates, Wildfire continuously executes several complex analytics queries over the latest data, both to summarize it and to trigger events such as showing new promotions to users. The summarization queries, detailed in Section 3.2, are displayed in the lower two panes of the demo display. The query determining which promotions to show to users, ShowAds, is far more complex. It joins Ads having remaining budget (indexed in memory) to tables Ads and Stores, then grids each Ad by its Store's location and Ad's category that define a key for the Ad on this grid. The current UserLocation (another in-memory index) and UserInterests for which Ads haven't been shown recently (determined using a third index) define the lookup key over this grid. Each eligible Ad is then scored as a function of the precise distance from its Store to the current UserLocation and a weighting of the UserInterest categories. This function simulates the output from a machine learning model that would be continuously trained using earlier Purchases, but implementation of that model training is future work.

3.1.4 *Setup*

The demo runs the Wildfire prototype on three remote 2-socket servers with 14-core Intel Xeon E5-2683 processors (2GHz) with 384GB of RAM. The operating system is Ubuntu 15.04 with Linux kernel 3.19.0-30. Each node stores the data on, and reads it from, an Intel 750 Series SSD.

Each node runs one instance of our Wildfire engine and a Spark Executor, as shown in Figure 1. The application driver runs on one of the three nodes; it issues OLAP requests to, and combines the results from, the Spark Executor on each node. The data for ingestion is generated and inserted locally on each individual node. The demo user-interface is accessed through a web-browser, since the nodes are remote.

We pre-populate the `Users`, `Stores`, `Ads`, and `UserInterests` tables of the application (see Figure 2). These tables do not change after initial population, as Section 3.1 describes. The `UserLocations`, `AdsShown`, and `Purchases` tables grow as the demo runs.

3.2 What the Audience Sees

The demo screen is composed of four major panes, as shown in Figure 3. The upper half of the screen illustrates the ingest capabilities of Wildfire. The upper-left pane reports the aggregated ingest throughput of the `UserLocations` table, as well as the throughput per node, every second. The cardinalities of each table are shown in the upper-right pane. The last four rows in this pane indicate the cardinalities for the four pre-populated tables whose cardinalities do not change during the demo. The first three rows in this pane give the cardinalities of the tables whose cardinalities do change due to the insertion of new rows. These cardinalities are continuously combined from in-memory counts maintained on each node, and are updated every second on the demo screen.

The lower two panes of the display are dedicated to reporting summarization (analytics) queries that are run continuously over all the persisted data, including the recently ingested data, while concurrently running the ingest queries. The demo viewer may choose which query to run in each pane. Each such query is a *GROUP BY* query that either performs a *SUM* or *AVERAGE*, grouping the data by product categories (lower-left pane, displayed as a bar graph) or by U.S. county (lower-right pane, displayed as a heat map). For example, the lower-left pane summarizes how the *remaining ad budget*, *total sales (purchases)*, *total cost of ads*, etc. change per product category as we insert new data. From movements in the height of these bars, the demo viewer can readily discern this continuous analytics update using new data, when high rates of ingest occur, and how long it takes to (re-)run that query. The lower-right pane of the screen has a heat map to indicate the rate of *purchases* or *remaining ad budget* per county. Positioning the cursor on a state causes a pop-up that enumerates each county and the exact *purchase total or remaining ad budget* per county, as shown in Figure 3 for the state of Utah.

At the bottom of the display, a narrow pane permits the demo viewer to adjust the system parameters, such as the frequency with which new `UserLocations` are generated for each `User`, the frequency with which `Ads` are shown to `Users`, and the click-through percentage for `Purchases`.

We start our demo with a low ingest rate and then gradually increase this rate. This helps in observing how the insert throughput of the system increases, how dramatically the table cardinalities grow, and the results reported by the OLAP queries change under varying ingest pressure.

After demonstrating how the system behaves under different ingest rates, we will let the audience interact with our user interface if they wish. They can change the ingest rates further from the bottom of the screen, or choose different analytics queries to run from the drop-down menu at the top of the bar graph and heat map panes. Furthermore, they can restart the cluster services with or without resetting table data using the *Reset* or *Restart* buttons, respectively.

4. CONCLUSIONS

The Wildfire system demonstrates high-rate inserts (up to 20 million inserts per second on just 3 nodes) with concurrently executing analytics queries. The demo shows a Wildfire engine connected to a Spark Executor on each node to issue analytics queries through Spark's Data Sources API. The connection to Spark exposes the analytics capabilities of Wildfire to the entire Spark ecosystem, including Spark SQL, graph processing, and machine learning.

This demo is a snapshot of work in progress. We are working on extending its integration into Spark SQL, particularly on extensions to Spark's Catalyst optimizer to perform more complex push-down analysis, and generating compensation plans for the remaining portions of the analytics queries that are not pushed down into Wildfire. In addition, we are exposing the OLTP interface of the Wildfire engine to Spark, so that applications running inside Spark have access to the full HTAP functionality. Lastly, we are adding ACID transactions to the Wildfire engine.

5. REFERENCES

[1] Apache Parquet. https://parquet.apache.org/.

[2] Apache Spark. http://spark.apache.org/.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[4] 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.

[5] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory Efficient Hash Joins. *PVLDB*, 8(4):353–364, 2014.

[6] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[7] 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), 2012.

[8] 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.

[9] 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.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.