On Mixing High-Speed Updates and In-Memory Queries
A Big-Data Architecture for Real-time Analytics

Tao Zhong  Kshitij A Doshi  Xi Tang  Ting Lou  Zhongyan Lu  Hong Li
Software and Services Group, Intel
{tao.t.zhong, kshitij.a.doshi, xi.tang, ting.lou, zhongyan.lu, hong.li}@intel.com

Abstract—Up-to-date business intelligence has become a critical differentiator for the modern data-driven highly engaged enterprise. It requires rapid integration of new information on a continuous basis for subsequent analyses. ETL-based and traditionally batch-processing oriented methods of absorbing changes into a relational database schema take time, and are therefore incompatible with very low-latency demands of real-time analytics. Instead, in-memory clustered stores that employ tunable consistency mechanisms are becoming attractive since they dispense with the need to transform and transit data between storage layouts and tiers.

When data is updated infrequently, in-memory approaches such as RDD transformations in Spark can suffice, but as updates become frequent, such in-memory approaches need to be extended to support dynamic datasets. This paper describes a few key additional requirements that result from having to support in-memory processing of data while updates proceed concurrently. The paper describes Real-time Analytics Foundation (RAF), an architecture to meet the new requirements.

Performance of an early implementation of RAF is also described: for an unaudited TPC-H derived workload, RAF shows a node-to-node scaling ratio of 88% at 8 nodes, and for a query equivalent to Q6 in the TPC-H set, RAF is able to show 9x improvement over that of Hive-Hadoop. The paper also describes two RAF based solutions that are being put together by two independent software vendors in China.

Index Terms—Big Data, Real-time, Low-latency, Analytics, Resilient Distributed Datasets, CRUD, Clustering.

I. INTRODUCTION

Operational analytics enable a company or an organization to analyze data from its day to day operations, including up to the minute transactions, so that it may act upon findings instantly. It is gaining momentum as institutions and individuals become pervasively data driven in all spheres of life. In addition to capitalizing upon well-catalogued historical knowledge, establishments are turning towards just-in-time analysis of information in motion that may just be seconds old, and not yet well categorized or linked to other data. A few examples of value from operational analytics follow:

- GPS-based navigation equipped with a static information base of road networks is a great modern convenience. Add instantaneous analysis of traffic conditions and it can guide motorists away or out of traffic logjams, to reduce wasted energy, accidents, delays and emergencies.
- A credit card company may use instantaneous correlation between a user’s transactions in order to detect and intercept suspicious transactions – such as, a transaction that breaks a pattern, or issues from a merchant that is too far away from the location of a recent transaction.
- A metropolitan or regional power grid which processes millions of sensor readings every second to modulate power generation, perform load-balancing, direct repair actions, and take policy enforcement steps. In addition to triggering immediate reactive actions, the data may also be used in long range capacity planning.

An essential feature in the above examples is the need to integrate new transactions into analysis results within a very short time—sometimes as short as a few tens of milliseconds. A second feature is the need to complete analysis queries quickly. Producing answers swiftly requires parallel execution of queries and agile sharing of information across parallel tasks. To keep pace with the growth of processing demand and that in the volume of content to be analyzed, organizations have increasingly embraced scale-out clusters (Hadoop[1], HBase[2], MPP databases[9, etc.] where more machines can be added readily in order to boost throughput and capacity. High latencies can make these clusters less suitable for operational analytics: transforming data between block format and in-memory formats, paging, and sharing via a distributed file system such as HDFS[4] prolong analysis.

To slash latencies by orders of magnitude, Spark [16], HP Vertica[14], and several other initiatives [10],[15],[7] propose maintaining data in memory (instead of demand paging it from disk), noting that rising capacities and dropping DRAM costs have made in-memory solutions practical in recent years. This paper identifies additional capabilities that are needed to blend in-memory data processing between transactional and analytic activities into a seamless whole. The resilient distributed datasets (RDDs) of Spark[16] make in-memory solutions less failure prone. In an architecture that we call Real-time Analytics Foundation (RAF), we propose enhancing the RDD approach so that resiliency is blended with a few additional characteristics listed next:

- Efficient allocation and control of memory resources
- Resilient update of information at much finer resolution
- Flexible and highly efficient concurrency control
- Replication and partitioning of data transparent to clients

Architecturally RAF elevates memory across an entire cluster to a first class storage entity and defines high level mechanisms by which applications on RAF can orchestrate distributed actions upon objects stored in cluster memory. By merging distributed memory and distributed computing, RAF delivers a platform on which applications designed for optimized in-memory execution can be distributed easily. Services that expose REST interfaces, Create-Retrieve-Update-Delete (CRUD) interfaces, SQL-like interfaces, or other object access methods for items which are in their in-memory formats can be created readily on top of RAF. To promote responsible and transparent use of memory, RAF opts to use a programming language such as C, C++, or mixed language environments in which garbage allocation is opaque as a general rule.

The paper is organized as follows. Section 2 sets out the motivation, explaining the requirements and how current solutions compare in delivering low-latency analytics. Section 3 then describes the overall architecture and implementation highlights that are the distinctive aspects of the solution. Performance of the current implementation is then brought out in section 4. Section 4 also describes two end user scenarios in which this approach is applied to meet the unique demands. Section 5 describes continuing and future work, and section 6 concludes.

II. MOTIVATION AND BACKGROUND

Data has a lot of value when mined. By analyzing an end-user’s recent browsing history, an artificial intelligence system
such as GoogleNow can automatically determine viewing preferences and push selections at him/her without explicit requests being made. A manufacturer or a retailer can extract and correlate consumer information streaming in from shopping outlets and use it in order to group and price products for a mix of high turnover and profitability. Smart deployment of police and ambulance services based on an analysis of traffic patterns can save lives during emergencies.

This understanding, that data has value, has led to the practice of mining it at the point of generation, for example, by subjecting it to streaming analytics. SQLstream[13] in effect runs queries on streams rather than on previously captured and ETL’ed data. However, as data continues to compound at brisk rates, institutions need to grapple with two broad demands – (a) accumulating, processing, synopsizing and utilizing information in a timely manner, and (b) storing the refined data resiliently, and keeping it accessible at high speed. Recognizing that large scale parallelism is critical and that the storage and computational capacities of single node multi-core systems quickly limit a solution, recent years have seen the emergence of softer-consistency, non-relational, cluster-based approaches for data processing described collectively as Big Data. The term Big Data itself is elastic and serves well as a description of the scale or volume of these solutions, but does not define a constraining principle for organizing storage or orchestrating processing. It is therefore useful to outline a few requirements for low-latency and high throughput analytics on datasets that are also subject to non-trivial volumes of updates.

1. **In-memory structures and storage:** As the complexity of queries increases, it becomes necessary not just to keep larger amounts of data in memory for avoiding disk latencies, but also to use data processing techniques that avoid having to marshal pointer linked and deeply nested structures between one software module and another, particularly when the modules are sharing data on the same node.

2. **Resiliency:** At the same time, it is obviously critical that data that must survive abrupt restarts of individual nodes is replicated across nodes and between storage tiers, in a timely fashion.

3. **Sharing data through memory:** Passing data by reference should be supported efficiently so that needless copying can be avoided. A module that produces data should be able to collaborate with a module that consumes data without the overhead of data copying, marshaling, and that of superfluous allocation or deallocation. Furthermore, it is important that modules that can form a chain of processing are able to operate on data flowing from a previous stage of processing into a next stage of processing while the output of the previous stage is in memory.

4. **Uniform interaction with storage:** Some systems designed for in-memory analytics split transactional object management from that for the analytics subsystem. Most systems mandate differing data storage and processing formats. Due to the resulting fragmentation of formats, developers and end users face the danger of being locked into specific approaches that later fail to meet shifting demands. At the same time, it is necessary to grapple with resilience questions that arise with distribution, and naming and storing of data that is not limited to a single “native” format. In a nutshell, solutions need to be interoperable and scalable, across multiple hardware platforms and software environments. The resulting impetus for OpenStack[17] has identified the need to move away from centralized databases and towards distributed object storage with object granular capabilities for locating, replicating, and migrating contents (see figure 1). To meet these objectives in concert with requirement #1, objects stored in memory should be interoperable, relocatable, replaceable in the same way that objects are placed in a block storage medium.

5. **Minimizing memory recycling:** Garbage collected language runtimes (such as Java) popular in many Big Data implementations liberate programmers from having to manage memory explicitly. However, the lack of direct control over memory deallocation can lead to performance overhead of garbage collection and cause problems such as long stalls during a stop-the-world garbage collection. More explicit memory management such as that in C and C++ is desirable from the standpoint of ensuring high performance.

6. **Efficient integration of CRUD:** In an operational analytics solution, it is important to be able to make small updates on a continuous basis, than to have to buffer up many changes and risk a long latency and resource intensive merging of those changes into the source from time to time. We borrow the acronym **CRUD** (Create/ Retrieve/ Update/ Delete) from [3] to refer succinctly to two capabilities: (a) writers perform their updates atomically at a granularity consistent with an application’s data model, and (b) readers have a stable, consistent view of the dataset. To minimize overhead and maximize concurrency, it is essential to support efficient CRUD operations on memory as well as disk versions of a dataset which is simultaneously accessed for analytics queries. In effect, without burdening the application logic, the datastore should provide a logically versioned view in which writers update contents granularity finer than that of a full dataset.

7. **Synchronizing efficiently:** Having to wait for milliseconds in order to achieve serialized execution across a cluster negates much of the benefit of in-memory processing. If synchronization is lock-based, then it requires distributed deadlock detection and guarding against node failures. While partitioning of data across nodes can mitigate distributed synchronization, local synchronization within a node can also be a serious performance inhibitor. Given the need for large scale concurrency—within and across nodes, it is essential to choose processing approaches that are implicitly data parallel.

8. **Searching Efficiently:** Many NoSQL solutions, particularly key-value stores that are available today sort records for
efficient searches based on record keys. Efficient searching also needs to be available across non-key attributes.

Many a distributed data processing alternative has emerged in the past decade to displace some previous relational database based solution. Several in-memory non-relational systems such as [6][10][16] have similarly emerged either as alternatives or as complementary approaches to in-memory relational databases. Let us briefly note the strengths, and the areas of improvement, among these four in-memory data solutions that are very popular today—Spark [16], Oracle Coherence[6], Redis[10], and Memcached[5].

Spark introduces an innovative distributed computing model that is particularly well suited to large scale batch processing. A complex analysis task is decomposed into a sequence of transformations, scheduled such that datasets generated by one transformation are used in another transformation with rarely a need to force the datasets out of memory into disks. Spark is however dataset oriented in that it is not well-suited to operate on a single record based on a record’s key, or to weave in small granularity updates into transformations. Redis is very popular due to its performance and ANSI C implementation, rich data types and consistency guarantees. It lacks a distributed framework for assembling complex, parallel computations. A single threaded model constrains its aggregation performance. Oracle Coherence is a mature commercial data grid solution with high availability, high scalability, rich data processing model, and support for transactions. That it is implemented in Java exposes it to garbage collection inefficiencies. And its query capability is constrained to filtering and to computing aggregations such as MAX, MIN and several others. Memcached, due to its stability and simplicity, is widely used for object caching; but that simplicity also precludes creating complex enterprise software – which limits the use of Memcached primarily as a backend assist for use in accelerating web servers. The next section discusses how to build upon the strengths and address some of the soft spots of the current popular solutions in order to better address the needs outlined in this section.

III. ARCHITECTURAL CONCEPTS AND IMPLEMENTATION

HIGHLIGHTS

The framework targets common user scenarios in which complex queries need to be conducted at very low latencies. Information upon which the queries operate may be available on some storage medium or generated dynamically on the fly as a result of ongoing transactional activities. In this section we translate the eight requirements articulated in section 2 into five design elements: (a) C and C++ based programming for efficient sharing of data through memory (b) Resilient storing of new content, (c) Efficient concurrency, (d) Processing information in motion, and (e) Fast, general, ad-hoc searches.

Transactional and analytic applications run as services, and end-user clients interact with these services --using interfaces that abstract away the details of how objects are distributed, cached, stored, replicated, partitioned, and so on, from the clients. For these applications, the RAF provides a distributed computing environment. The distributed computing environment is integrated with a memory-centric, distributed, storage system; that is, one application can pass a data handle to another in order to share data in memory with the latter, without first having to serialize or deserialize data for storing to file systems or transmitting via network connections.

To outline the RAF framework, it is useful to discuss four concepts, Delegate, Filter, RDD, and Transformations. We do this with respect to figure 2. In figure 2, new data or updates to existing data occur as a result of clients contacting a Storage Service. Typically the updates will happen first in memory, and, sometimes, the updates may become propagated to non-volatile media.

- An RDD is an acronym for a Resilient Distributed Dataset, introduced in [16]. RAF uses the RDD construct for reasons identical to those in [16], viz., resilient storing of information in memory of one or more machines, together with assurance that in case of failure of one or more machines, an RDD can be reconstructed from precursors by repeating well-formed operations that produced it.
- A Transformation is an operation applied to one or more RDDs in order to generate a new dataset or a result that may be transient. The concept of transformations is similar to that in [18]; and RAF transformations include such transformations as join, map, union, etc.
- A Filter is a particular type of transformation. A filter, as in [18] produces, out of an input dataset, a resulting dataset whose contents satisfy a specified condition.
- In order to facilitate efficient sharing of both volatile and non-volatile storage between applications that need to make updates and applications that need to read the data, we create a level of indirection to the storage. This indirection is obtained through a set of wrapper functions that ensure a stable view of content to read-only consumers, even as content mutates. This is described further in §3.1.

Thus in figure 2 raw information – whether it’s preloaded into in-memory storage from disks or dynamically generated by data producing clients, is first made stable through the use of Delegate modules, and then filtered in order to create initial RDDs (such as RDD1 and RDD2 in the figure). Chains of Transformations then produce the desired analytics results. To reiterate, data sharing is through memory instead of as opposed to a distributed file system such as HDFS. This is supplemented by a flexible execution model in which applications interact with other applications synchronously (i.e., using RPC), or asynchronously via queued messages. High performance analytics applications can remain loosely coupled (and therefore efficient) by using message queues.

A. Efficient Storage Sharing using DELEGATE

A shared CRUD[3] data source (§2) furnishes different
information to queries at different points in time. This introduces an impedance mismatch for software that operates on RDDs since a typical storage interface for implementing CRUD actions does not have the same breadth of operations that an RDD possesses. This difficulty is removed in RAF by inserting a bridge module named DELEGATE. The purpose of DELEGATE is to create a version of the datastore at a particular time, and present it as a memory resident RDD. Beneath the DELEGATE module, CRUD operations can proceed directly against the mutable datastore. While analysis operations proceed against the RDD that is created on top of the view produced by DELEGATE, some entities may be added or deleted without affecting the analysis.

For efficiency, the DELEGATE module employs copy-on-write through pointer indirection. DELEGATE module provides C/C++ wrappers for objects in storage — whether persistent or volatile, and using these wrappers, an RDD creating operation can share memory efficiently but safely with concurrent transactional operations that may change the composition of a dataset. While embracing the concept of RDDs from [16], RAF deviates in implementation of RDDs from [16] in order to carry out explicit instantiation and sharing of RDDs and facilitating coordination through DELEGATE modules, ensuring that data is directly and efficiently accessible from CPUs for analysis computations.

**B. Memory-centric Storage Operation**

For driving very low latency analytics applications on RAF maintain data in memory, and share access to it efficiently. As machines with large memory sizes have become more common in recent years, accumulating and operating upon large amounts of data has become practical. With a shift to softer and application managed consistency — away from strict ACID semantics in the data layer itself, it is possible for applications to spread data among a cluster of large memory machines to further expand solution sizes. Applications also control which data needs to be flushed to a non-volatile medium and when; and this is achieved in RAF by means of disk or file system plugins.

**3.2.1 Reliability**

It is necessary to address the concern that volatility (of main memory) can compromise reliability and availability. An RDD is recoverable by design. Thus it is the data source from which an initial RDD is obtained (via Delegate, as described in §3.1) that needs to be recoverable even if the primary mode of committing changes to a non-volatile copy of the source, such as to a disk is non-synchronous. This is achieved as follows.

Applications run on a cluster and can query the system to know how many nodes are present in the cluster. Thus an application can write an update to more than one node; and if the cluster size drops to 1, can record changes to a non-volatile medium. A storage structure (such as a file) is divided into uniformly sized partitions with records distributed among partitions through hashing of record keys. A partition is typically copied to at least one other back up node, and the storage layer maintains the mapping between each partition and the additional nodes where that partition’s backup copy is maintained.

- **When a node N is about to leave a cluster, or if it leaves the cluster abruptly, the storage layer discovers which primary and backup partitions are on N. For each of N’s primary partitions, it selects one out of its backup partitions and marks it as primary, and creates a new backup partition on yet another node in the cluster. For any backup partition on N, RAF creates a new instance of the backup partition on some other node.**

- If a new node X joins the cluster, then RAF rebalances partitions across all nodes including X, in background.

Updates to a partition are automatically streamed to a partition’s replicas through a network interface, without burdening the application logic. Applications can do control the replica count as well as whether replication of a write is required to be synchronous.

**C. Data and Storage Types**

In this subsection we briefly describe application visible aspects of data layout and distribution. §3.3.1 explains how RAF implements data types, and §3.3.2 on storage types describes two concepts that relate to how data is distributed across nodes.

**3.3.1 Structured Data**

Structured data is expressed by a two part definition that is illustrated in Figure 3. On the right is a protocol buffer [8] that shows two types: CustomerKey, and Customer. As described in [8] this structure is described in a file which is compiled in order to yield data accessor methods that are very efficient in parsing the necessary fields from a message stream. In addition, a metadata structure shown to the left in figure 3 also in JSON format defines tuples, by describing four aspects: (1) the names (analogous to names of relations in a database), (2) the key attributes, (3) non-key (or “value”) attributes, and (4) a store type, which we will describe a little further in the next subsection. In implementing this type of extensible structure, RAF supports both memory and disk based, flexible organization of structured data, with efficient serialization [8]. In particular, the responsibility for locating the data — whether it is in memory, on disk, local, or remote is offloaded from applications and absorbed into the RAF platform modules. Applications supply the metadata and protobuf definitions in a file, and the RAF framework automatically creates the necessary internal structures and serialization necessary for transmitting or receiving data across node boundaries.

![Figure 3: Type and Store Metadata](image)

While the use of Protobuf as an efficient exchange format among distributed entities is well understood, its use in the RAF is particularly advantageous because real-time analytics is built on the proposition of propagating updates resulting from CRUD actions as quickly and efficiently as possible. This is best seen by considering row- and column-based alternatives for database records: in row-based structures, updates would require extensive parsing, while column-based structures are not well suited for high rates of updates.

**3.3.2 Storage Types:**

RAF provides two attributes for datastores: replicated store and partitioned store. A datastore may be single or partitioned, and it may be optionally replicated. When a datastore is small, and is updated rarely but is read frequently, it is advantageous
to keep it all of it as one single extent (preferably also in memory) at one home location, but make one or more replicas of it available on other nodes. This is captured as case (a) in Figure 4. If on the other hand, it is written frequently enough (case (c) in Figure 4), then it makes sense not to replicate the single extent that hosts it, widely. For high reliability, at least one copy of it would be advisable on a remote node as backup. If a datastore is large in size (case (b) in Figure 4), then it is divided into multiple extents or partitions; and each partition is given at least one backup copy on some other node other than its home location.

![Figure 4: Attributes of Storage](image)

When there are backup copies, or if there are replicas of a partition, a master copy of any given partition is kept on some node. That node is the one which storage operations ("CRUD" operations) contact and the master copy reflects the changes from the update operation into remote copies. This is done in accordance with application guided consistency philosophy. An application can specify to the RAF whether it wants write-through updates or wants remote copies performed using a write-behind approach. Under write-through, the update operation returns control to application once data is updated in all the copies; and under write-behind, the operation returns as soon as it completes locally on the master copy, and the changes are reflected to other nodes asynchronously.

### 3.3.3 Storage Service Interface

For CRUD operations, RAF furnishes both a command line interface and a C++ language application programming interface. As described in §3.3.1, protocol buffers are used for accessing data, which makes it easy to implement programming interfaces in other languages. The following very simple grammar is used to write command line scripts:

```
COMMAND ::= [SVC_NAME OP_NAME PARAM EXA_PARAM] ";" \\
SVC_NAME ::= ID \\
OP_NAME ::= ID \\
PARAM ::= JavaScript Object Notation (JSON) \\
EXA_PARAM ::= ID "=" PARAM \\
ID ::= [a-zA-Z0-9-_\.]+ \\
```

The following, for example, creates a new customer “Tom5” in a business directory in a storage dataset named “Customer”:

```javascript
store.service.insert("store name":"Customer", key="c_customer":"2340055", c_name:"Tom5", c_nationkey:15, c_phone:15000000001, c_city:"Beijing"; \\
```

D. Distributed Execution of Analytics Tasks

As described earlier in this section RAF reuses the concept of resilient distributed datasets (RDDs) from [16] in order to capitalize on memory resident data for low-latency analytics. And with the introduction of Delegate (§3.1), RAF also weaves in CRUD capable stores as source datasets from which RDDs can be derived – just as [16] shows how datasets such as HDFS files can be used for deriving RDDs. Successive stages of transformation described by a directed acyclic graph (DAG) is used, as in [16], yield a memory-based, low latency computation plan for analyzing data in mutable/CRUD storage to produce query results. By dividing very large datasets into multiple partitions each of which is backed-up by a copy allows this approach to scale and remain resilient in spite of moving from a read-only to read-write storage model.

RAF adopts the SEDA [12] model to achieve highly efficient, composable parallelism. Efficiency results from event-based dispatching of run-to-completion methods which minimize context switching for synchronization. Memory-based storage is particularly synergistic since it drives down the probability of wait-states that might otherwise result from having to page data into memory from disk-based storage. Each node acts as a monitor over data partitions local to it, and each operation that would operate on that data partition is queued. With increasing core counts driving up the number of computing elements within each node and with larger and larger node count clusters becoming commonplace, this architectural approach is well positioned to match low-latency in-memory execution with non-blocking, data flow form of execution.

### 3.4.1 Analytics Tasks Interface

The storage service interface described in §3.3.3 is complemented by a service interface through which clients can compose and execute a DAG of operations to orchestrate analytics tasks. A command can use the keyword `in_r_dd` to designate each source operand in a filter/transform operation, and the keyword `out_r_dd` to name the result of the operation. An RDD may be described as an `in_r_dd` in one operation, and as the `out_r_dd` of some other operation, which makes it possible for an analytics client to construct a pipeline of operations through which designated flows of data occur.

The syntax of an analytics command differs in minor respects from that of a storage service command. The grammar for an analytics task, which we refer to as an RDD service operation, is the same as that described in §3.3.3 except that the JSON content is different for each OP_NAME (i.e., by operation). Let us use an example to illustrate. Following is a command which takes as input an RDD named `customer`, and then applies a filtering operation to extract a subset of customers who are in the city of `Beijing`, and names the resulting RDD `bj_customer`:

```
   j_customer = \{ \\
     "transform": "filter", \\
     "in_r_dd": "customer", \\
     "out_r_dd": "bj_customer", \\
     "filter": \{"c_city": "Beijing"\} \\
   \} \\
```

The following example further shows (a) how DAGs can be specified step by step and (b) how independent tasks that happen to share common intermediate datasets can reuse, instead of regenerating the shared datasets. Let us say we have the following two independent queries – (1) : Compute average sales per customer for customers that are from either Beijing or Shanghai, (2) For customers from Beijing, identify
the subset of customers who have purchased more than once, and then for that subset, calculate the average sales. The high level depiction is as below:

**Computation Flow X:**

- **Customers**
- **Sales Reports**
- **Customers from Beijing**
- **Customers from Shanghai**
- **Customers from Beijing or Shanghai**
- **Repeat customers from Beijing**
- **Repeat customers from Shanghai**
- **Average sales per customer**

**Computation Flow Y:**

- **Customers**
- **Sales Reports**
- **Customers from Beijing**
- **Customers from Shanghai**
- **Customers from Beijing or Shanghai**
- **Repeat customers from Beijing**
- **Repeat customers from Shanghai**
- **Average sales per customer**

Both flows take as input a sales reports file, and progressively compute the RDDs to arrive at two different targets: the first flow is aimed at taking all customers who are from either Beijing or Shanghai, and averaging the sales in that slice of customers, while the second flow aims to calculate the average sales per repeat customer in Beijing.

In RAF, the above computation could be organized as follows:

**Common Precursor Action** (across X and Y, above):

- **Customers**
- **Sales Reports**
- **Customers from Beijing**
- **Customers from Shanghai**
- **Customers from Beijing or Shanghai**
- **Repeat customers from Beijing**
- **Repeat customers from Shanghai**
- **Average sales per customer**

In order to specify the above compositions flexibly, applications would proceed to specify the following three computations to an in-memory RDD service, as follows:

```java
/** --- extract the dataset "customers from Beijing" ---
 * rdd.service.create("transform_type":"filter",
 * "in_rdd":{"rdd_name":"customer"},
 * "out_rdd":{"rdd_name":"bj_customer"})
 * transform=[["op":"EQ","sub_expr":{"op":"FIELD","param":"c_city"},
 * {"op":"CONST","param":"Beijing"}]}
 */

/** --- extract the dataset "customers from Shanghai" ---
 * rdd.service.create("transform_type":"filter",
 * "in_rdd":{"rdd_name":"customer"},
 * "out_rdd":{"rdd_name":"sh_customer"})
 * transform=[["op":"EQ","sub_expr":{"op":"FIELD","param":"c_city"},
 * {"op":"CONST","param":"Shanghai"}]}
 */

/** --- union Beijing and Shanghai Customers ---
 * rdd.service.create("transform_type":"union",
 * "in_rdd":[{"rdd_name":"bj_customer"}],
 * "out_rdd":{"rdd_name":"bj_sh_customer"})
 */

/** --- compute avg. sales across Beijing & Shanghai customers ---
 * rdd.service.action("action_type":"average",
 * "rdd_name":"bj_sh_customer"}) action =
 * {"expr":{"op":"FIELD","param":"sales_amount"}}
 */

All the above script commands follow the grammar in §3.4.1. The name for the analytics service is rdd.service. Keywords create and action respectively denote whether in the above example rdd.service is being asked to create an RDD (tasks a, b, and c) or compute a result (tasks d, e) by operating on one or more existing RDDs. When there is only one RDD to operate on in the case of an action (as in tasks d, e), it is permissible to drop the in_rdd keyword described in §3.4.1. RDDs to operate on are identified by the keyword rdd_name in both types of operations. The use of rdd_name keyword to name an RDD explicitly in this way allows an application to avoid creating redundant RDDs. For example, suppose one process A executes the actions shown in computation flow X; that process would create RDDs bj_customer and sh_customer. Another process B that executes the actions in computation flow Y would end up repeating the computation of bj_customer for its query. With the RDDs explicitly identified by rdd_name keyword, flow Y can avoid recreating bj_customer if A has already begun creating it and vice versa.

We can represent the overall flow of both computations by the DAG in Figure 5. Let us use this diagram to discuss how the execution would occur using the RAF. In the DAG of Fig. 5, each box is an RDD. Solid lines represent RDD creation/transformation steps, and dashed lines represent action steps. Thus bj_customer and sh_customer are created by filter transformer on the customer dataset, bj_sh_customer is created by a union transformer, and bj_customer_2 is the

![Figure 5](image1)

![Figure 6](image2)

![Figure 7](image3)
result of filtering for repeat customers in bj_customer dataset. Averaging actions on bj_customer_2 and bj_sh_customer respectively yield the two results from two queries. As noted above, RDD bj_customer is used in both queries and its creation is shared (not repeated) in the execution of the DAG.

Figures 6 and 7 show how partitions (§3.3.2) work. Suppose customers were a very large dataset that is partitioned across three nodes—shown respectively by blue rectangles numbered 1,2,3. The number of partitions, incidentally, is controllable by user. Then, as shown in Figure 6, the creation of derivative RDDs— bj_customer and sh_customer can also proceed partition by partition; as would the creation of the union RDD bj_sh_customer. Thus data decomposition in source RDDs would be easily carried over into destination RDDs. This allows, as shown in Figure 7, keeping computations node local by maintaining corresponding partitions in the same node where the semantics of an operation allow. One counter example where such an optimization may not be applicable is a join that requires cross-node communication. However, this is made more efficient in RAF by replicating RDDs that are rarely updated (§3.3.2) but which may be read many times in operations such as joins. Finally, reliability in such operations is obtained by virtue of RDDs being recomputable as described in [16]. In RAF, the maintaining of backup partitions further improves availability of RDDs.

IV. RESULTS

In this section we share the performance readout for RAF, using unit tests that demonstrate outstanding transactional and decision support performance. We do this in section 4.1. Then in section 4.2, we outline two RAF-based applications from established commercial software vendors, and describe their performance, to portray how straightforward it is to create realistic operational analytics solutions atop RAF.

A. Unit Testing:

We measured RAF on two fronts: how well update operations scale, which represents memory-centric storage operations performance, and, how long does it take to complete a query to show distributed analytics performance. Table 1 has configuration data for the update test. For update operations, we borrow SSB schema from Star Schema Benchmark [19] and utilize SSB-DBGEN tool to generate data file. Then the update testing parses the data file and insert all the records into RAF. The performance metric is measured by TPS, which is total record number divided by total execution time in second.

Figure 8 shows consistent increase in the throughput of this test as it distributes over multiple nodes; with an aggregate scaling of 7.0x for an 8 node cluster consisting of Intel® Xeon® E5-2680 processors. In this test, for the 1 node test (with asynchronous backup copying to other nodes) we obtained an average CPU utilization of about 80%. Despite it being an update only test (and therefore network intensive), the percent of time spent in operating system was only 7%, which is reflective of the fast data capture, distribution, and storage capability of RAF.

<table>
<thead>
<tr>
<th>Server</th>
<th>Intel® Xeon® E5-2680 (8C, 2.7GHz), 128GB DDR3-1333, 10Gb NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>10Gbps Switch</td>
</tr>
</tbody>
</table>

Table 1 Configuration

The second test uses a query that is logically identical to query 6 of the industry standard TPC-H benchmark, rewritten for execution on top of RAF, and for execution on top of Hive. Both the RAF test and Hive test are performed on an 8 node Intel® Xeon® E5-2680 cluster and the dataset size used is 120G. As shown in Figure 9, RAF achieves nearly 9X the performance of Hive on 1st Iteration. We do not compare 2nd and subsequent iterations of this test with Hive because, since RAF explicitly uses memory as storage, its response time for this query drops to 150ms but Hive cannot benefit in like manner. These unit testing results show the advantage of in-memory distributed processing oriented design of RAF, to be substantiated further by more sophisticated workloads. Currently we are building more TPC-H derived queries, and a high concurrency operational analytics workload.

B. Solution-level Implementation and Testing:

We describe in sections 4.2.1 and 4.2.2 two solutions being developed by software vendors and initial solution performance. They each require storing a large volume of new data at the same time that the information must be analyzed in real time. Both sections describe the problem being solved and the performance obtained by the solution in current testing.

4.2.1 Telecommunications Subscriber Management

RAF is used by an ISV (independent software vendor) whose customers are telecommunications service providers. The customers have hundreds of millions of subscribers. A “Unified Service Platform” (USP) provided by the ISV conducts subscriber transactions, many of which may be self-service transactions. The typical scenario is comprises a subscriber pushing fresh credits into his/her mobile services account, prepaying or paying on demand for a broad spectrum of products. A subscriber belongs to one province company of a telecommunications service provider, but he/she can use pre-
paid cards issued by province companies that are different from the one to which he/she belongs. USP performs transactions, routes requests and responses back and forth among participating IT systems of province companies, logs transaction histories, and carries out desired analytics against stored data or in-progress transactions.

For one specific customer (telecommunications provider), there are more than 300 million subscribers and USP transactions have a volume of about 10 million per day, with a peak volume of 1000 transactions/second. The customer has requested the ISV to furnish USP with two types of business services that need to use RAF as a single seamless platform for rapid integration of real-time analytics solutions. This paper presents RAF, an architectural approach that meshes memory-centric non-relational query processing for low latency analytics with memory-centric update processing to accommodate high volumes of updates and to surface those updates for inclusion into analytics. This blending is kept efficient by merging data management between the two spheres of operation by using a transparent versioning capability that we term Delegate, which participates as a special type of content transformer in a hierarchy of RDD transformations. In RAF, protocol buffers are used to obtain data abstraction and efficient conveyance among applications, providing applications with a high degree of independence in location, representation, and transmission of data. The use of protocol buffers is particularly valuable as it removes the need for producers and consumers to coordinate explicitly in partitioning, tiering, caching, or replication of information that may be arriving at high velocities. These improvements for distributing a memory-based storage service are combined with a message queues based execution partitioning mechanism in which each node acts as a local monitor over its data partitions. A light-weight but expressive interface makes it easy for RAF services to map various transformations that need to occur in the course of execution of a query into the data flows that need to occur among the distributed execution agents through message queues, thus hiding all of the plumbing from the services that need to use RAF as a single seamless platform for modifying as well as querying data in pooled memory and aggregate computing capacity of a cluster of machines. Using unit tests we show high cluster scaling capability for transactions, an order of magnitude latency improvement for query processing. The paper also describes two real-world usage scenarios in which RAF is being used to create high throughput operational analytics solutions. With rapidly increasing memory capacities and large, open-stack based clusters, RAF provides a collection of architectural techniques for rapid integration of real-time analytics solutions.

VI. CONCLUSION

This paper presents RAF, an architectural approach that meshes memory-centric non-relational query processing for low latency analytics with memory-centric update processing to accommodate high volumes of updates and to surface those updates for inclusion into analytics. This blending is kept efficient by merging data management between the two spheres of operation by using a transparent versioning capability that we term Delegate, which participates as a special type of content transformer in a hierarchy of RDD transformations. In RAF, protocol buffers are used to obtain data abstraction and efficient conveyance among applications, providing applications with a high degree of independence in location, representation, and transmission of data. The use of protocol buffers is particularly valuable as it removes the need for producers and consumers to coordinate explicitly in partitioning, tiering, caching, or replication of information that may be arriving at high velocities. These improvements for distributing a memory-based storage service are combined with a message queues based execution partitioning mechanism in which each node acts as a local monitor over its data partitions. A light-weight but expressive interface makes it easy for RAF services to map various transformations that need to occur in the course of execution of a query into the data flows that need to occur among the distributed execution agents through message queues, thus hiding all of the plumbing from the services that need to use RAF as a single seamless platform for modifying as well as querying data in pooled memory and aggregate computing capacity of a cluster of machines. Using unit tests we show high cluster scaling capability for transactions, an order of magnitude latency improvement for query processing. The paper also describes two real-world usage scenarios in which RAF is being used to create high throughput operational analytics solutions. With rapidly increasing memory capacities and large, open-stack based clusters, RAF provides a collection of architectural techniques for rapid integration of real-time analytics solutions.

REFERENCES