Pulsar – Real-time Analytics at Scale

Sharad Murthy Tony Ng Bhaven Avalani
Xinglang Wang Ken Wang Anand Gangadharan
eBay, Inc.

ABSTRACT

Mobile devices continue to power high growth in online commerce. Web-scale commerce platforms have to engage hundreds of millions of shoppers who choose from more than a billion items offered by tens of millions of merchants. In order to convert shoppers to buyers, it is critical to respond to their actions in real-time, offering them relevant next steps in their purchase journey.

Consumer expectations have grown rapidly during the past few years. Now, they expect in-session personalization across multiple screens, including hyper- relevant advertisements that use rich context aggregated from the sensors they carry.

Fraud detection, business activity monitoring, and so on have very similar needs to process data in near real time, glean actionable insights within seconds, and generate signals for immediate action. The latency inherent in a store-and-process model makes batch-oriented systems such as Hadoop unsuitable.

The sheer data volume and the low latency requirements demand in flight data processing instead of a store and process model as in batch oriented systems. The stream processing infrastructure needs to be distributed across data centers, yet have a programming model that does not require very specialized skills. The system must be highly available (i.e. allowing for new applications and application versions be deployed with zero downtime).

This paper discusses the design of **Pulsar**, a real-time analytics platform that is well suited to address the class of problems noted above. We will discuss design choices for critical subsystems and components along with the rationale.

1. INTRODUCTION

User behavior events contain structured information such as user-agent or native application identifier (IP address) Applications can extend to include attributes relevant to a specific event type. These events are captured by native or web applications for real-time and offline analysis. At eBay, we capture in the order of hundreds of thousands of events per second and growing fast as the number of gestures continue to increase, along with user growth. BOTs masquerading as humans generate a significant amount of this event traffic. BOTs are a source of valid (e.g. search engine crawler) and unproductive (crawlers that scrape pages for information) traffic. In either case, we need to protect resources of downstream systems by detecting them and quarantining those events or process them with different service levels. It's important to note that BOTs can be fairly sophisticated when trying to mimic human behavior, and are known to purchase items!

A class of data enrichment of user behavior events such as geo location lookup, device classification, demographics, and many others is best done as early as possible in the data pipeline to centralize processing logic at the cost of de-normalizing data a bit. In other scenarios, such as Risk and fraud analytics, the real-time user behavior patterns need to be augmented by historical summary information accumulated over time periods, extending to years. In such scenarios the system throughput demands require a very fast lookup that cannot be met by RDBMS.

Sessionization is a key construct that allow events to accumulate in a data container until a user session is deemed complete, usually based on inactivity for a period of time. Many business metrics including conversion attribution depend on the notion of a user-session, which is best done once early in the data pipeline.

In the area of business activity monitoring, there is a need for real-time aggregation of time-series metrics for exploratory visualization and for generating alerts based on random groupings of dimensions. This use case in particular requires even higher ingest rates that cannot be served by a RDBMS system. Metrics aggregation in real-time poses challenges, such as explosion of counters due to large number of dimensions and high cardinality dimensions with very sparse data.
In order to allow an ecosystem of applications to tap into a rich data stream, it is important to have the notion of partial views of original streams which filters, enriches, and mutates event data for lower cost processing downstream.

In this paper, we propose a design for a data pipeline: build a highly available distributed computation system with CEP engines for most of our computation, processed in real-time. Our proposal is to treat the event stream as a database table. SQL queries can be executed against real-time streams to create aggregates just like one would do against a database table. We describe how we can enrich, filter, mutate and create partial views of the event stream in flight.

2. DATA AND PROCESSING MODEL

2.1 User Behavior Data

User behavior data is unstructured or semi-structured for the most part. It is made up of many tuples of information. Each tuple represents a dimension. User behavior data stream is made up of a continuous sequence of events. Each event

contains contextual data around a click and is made up of many dimensions (up to 40 dimensions). For example, page ID, IP address, geo information like city, region, and country or device classification data like browser type, device type, and many more are examples of dimensions.

Many of these dimensions have high cardinality, making the data set sparse. One of the challenges of creating aggregates for a grouping of high cardinality dimensions in real-time is the explosion of counters. This is a very challenging problem to deal with in both space and time in a real-time computation environment.

It is common to see user behavior data streaming at rates exceeding millions of events per second. The events are generated in unpredictable patterns. The use cases that operate on this data demand very low latencies. This requires in-memory processing of data streams as the data is flowing through the system.

2.2 Data Enrichment

The user behavior streams contain both BOT and user behavior activity. Our end users are only interested in user behavior activity events.. This requires BOT activity events to be detected and filtered out. Some other use cases only process BOT events and for them the user activity events need to be filtered out.

Often other sources of data with very valuable information need to be combined with the user behavior stream. Geo information, device classification, demographics and segment data are examples of such data. The challenge is to design scalable data stores that can be queried at the rates being discussed. Many of the use cases under consideration require this type of data. It would be extremely expensive to have applications serving these use cases to independently access these data sources. It would be most efficient to provide capabilities to decorate the data streams with newer dimensions in-flight. This involves looking up a store using one of the dimensions in the event as a key. We can deploy strategies of caching other sources of data locally on the processing node or externally in a fast lookup cache. This decision will have to be based on how often the data changes and size of the data. This would be analogous to data base table joins.

2.3 Sessionization

Sessionization is a process of state management by grouping of related events identified by a common key. In web analytics, a visit or session is defined as a series of page requests or, in the case of tags, image requests from the same uniquely identified client. A visit is considered done when no requests have been recorded in some number of elapsed minutes. A 30 minute limit ("time out") is used by many analytics tools but can, in some tools, be changed to another number of minutes.

A session state typically involves one or more counters of certain activity in that session (e.g. recording of page navigation).

Our solution is designed to support tenant defined sessions. A tenant is a consumer of the sessionized data. Each tenant's session has its own lifecycle. A tenant's session is identified by a unique identifier made up of one or more dimensions from the events driving the sessionization. The session identifier, duration, metadata and state management logic are all specified through a declarative syntax.

2.4 User defined partial views of original streams

Typically, events sourced from applications have a large payload consisting of a large number of dimensions. Moving such large events over a distributed pipeline across a wide area network is very expensive. Most consumers are mostly interested in consuming partial views of the original stream. The consumers of these streams will have to make substantial investment to get a partial view of the original stream if they consumed the original stream. A subscription based capability needs to be provided for users to declaratively define partial views of the original stream. A user can define the dimensions in the original stream of interest to them and the filtering rules to be used to populate the view. The data populated in the view needs to be regulated based on a subscriber's authorization level.

2.5 Computing aggregates in real-time for groups of multiple dimensions

Aggregation is a process of producing summary data for a group of dimensions. RDBMS systems or PIG or HIVE provide aggregation functions like COUNT, SUM, MIN, MAX, AVG, DISTINCT COUNT, TOP N, QUANTILES and many more.

Consider the following statement in the RDBMS world `select count(*) as METRIC1, column1, column2 from SOMETABLE group by column1, column2.` This statement finds a count of occurrences of unique groupings of column1 and column2 in a table called SOMETABLE.

This query runs against a single database instance. In a sharded environment, an application would have to run this against each shard and then further aggregate the result set returned by each shard. The query can take a long time if the number of rows being scanned is very large.

Now consider executing a similar query in real time on live streams. The dimensions in the event will take the place of columns and the event stream replaces the table. In section 2.1 we mentioned that it is typical to be processing millions of events per second in Ecommerce systems. Real time data streams are continuously moving data unlike a database. We need to define the start and end point in the stream for the

aggregation function to process data in a continuously moving stream. We achieve this by defining a window of time over which the aggregation is performed. The window can either be a tumbling window or a rolling window. A sample SQL below shows a query applied to a data stream that counts unique groupings of dimensions D1 & D2 over a tumbling window of 10 seconds.

```
create context MCContext start @now
end pattern [timer:interval(10)];

context MCContext
insert into AGGREGATE select count(*)
as METRIC1, D1, D2 FROM RAWSTREAM
group by D1,D2 output snapshot when
terminated;

select * from AGGREGATE;
```

Since real time stream processing is all done in memory we are constrained by space. Since our streaming data is sparse, we have an explosion of counters putting pressure on our memory resources. Hence we propose to keep the window small (10 seconds). We propose a partitioning strategy to spread the work load across a cluster of computation nodes. The events are scheduled into the cluster using a consistent hashing algorithm. The algorithm fits a hash key on a logical ring to find the node to which the event needs to be scheduled. The hash key is created by composing a key from one or more event dimensions and computing a hash (H(D1, D2,..)) of the composed key. We propose to use a 128 bit hash function so we get a nice distribution of the events across the cluster.

The data produced through aggregation is time series data. Since all the aggregates are in memory there is potential to lose the aggregate if the node dies. Our strategy is to keep the aggregation windows short (30 seconds to 1 minute). When the window rolls, the aggregate is emitted as a metric event by the engine. We store this event in a time series data store so we can create aggregates over longer windows (1 hour or 1 day) in the data base. This snapshotting approach acts like a journal – we could store away the snapshot in a time series database and restore the aggregate on a different computation node to recover from a computation node failure if we decide to. Since most of our use cases are statistical in nature we have decided to live with the loss as it is very small.

We also use the snapshots to drive visualization widgets in real time. These widgets are hosted in real time dash boards and produce visualization of the data. Several real time visualizations have been produced in this manner in our production environment.

This approach of aggregation produces cubes over known dimensions. In such cases the grouping of dimensions are known upfront. It is very well suited for real time use cases. When the data is stored in a time series data store it can be used for reporting use cases. However this approach by itself is not suitable for data exploration. Data exploration involves executing adhoc queries to create aggregates grouping random dimensions over data spanning many years. In section 6 of this paper we will explore an approach to combine online with offline to show how we can take advantage of pre-aggregation to contain cost and improve query response of a time series data store.

2.6 Computing Top N, Percentiles. and Distinct Count Metrics in real time

Consider the following query for finding the top 10 groupings of dimensions D1, D2 & D3 computed over a 1 minute window

```
create context MCContext start @now
end pattern [timer:interval(60)];

context MCContext

insert into TOPITEMS select count(*)
as totalCount, D1, D2, D3 from
RawEventStream group by D1, D2, D3
order by count(*) limit 10;

select * from TOPITEMS;
```

This query execution cost is proportional to the number of unique combinations of D1,D2,D3 and the rate of event arrival into the window. If one or more of these dimensions has high cardinality then the number of counters will explode putting pressure on memory resources. The ORDER BY operator will cause sorting of the data when the window rolls. This will consume a lot of compute resources putting pressure on latencies. It could also cause frequent garbage collection in our processing application which are built in java. Other metrics like finding the distinct count and percentiles are very useful but have similar issues as they are memory intensive. This makes it very difficult to execute these types of queries in real time at our volumes.

However, if we consider using approximation algorithms for computing top N, percentiles and distinct count queries, then we can operate within a fixed amount of memory that we can control. We propose to use frequency estimation techniques for computing top N. In this approach we only keep frequently seen groupings in memory. When the storage becomes full, the least frequently used grouping is discarded from memory to make way for a new count. We have implemented this algorithm as a special aggregate function that can be used in a query as shown below.

```
select TopN(1000, 10, D1) as
topItems from RawEvent();
```

The first argument to the aggregate function specifies the capacity, second argument corresponds to the max items to return (top N) and the last argument is the dimension from the event.
Our proposal uses HyperLogLog[9] and TDigest[10] algorithms for computing distinct count and percentiles. We provide aggregate functions for these two computations which can be used in a SQL on an event stream.

With this approach we can control both space and time for the computation at the cost of accuracy. In our experience we find that these approximate algorithms introduce an error of up to 1 percent depending on the data set and the capacity allocated to the algorithm. This is acceptable for most statistical problems in our opinion

2.7 Dealing with out of order and delayed events

It is very difficult to guarantee order in a distributed system. Sometimes due to network issues we have to store events in persistent queues and replay them later. This causes delay and might impact order.

Mobile devices typically batch events and will send a batch every minute. The events could travel through intermediate queues and arrive much later than when the activity was recorded on the mobile device. Typically we are required to compute metrics grouped by dimensions across devices in a specific window. We want our metrics to be as accurate as possible to the nearest minute, hour or day.

In order to compensate for out of order and delayed events we propose that the data sources sending events in to our system decorate the events with a time stamp as one of the dimensions. In our system the time stamp will be rounded to the nearest minute and injected back into the event. We propose that all the metrics have the rounded timestamp as one of the dimensions in the grouping. When the event comes delayed or out of order we are able to attribute it to the correct rollup window in the time series database.

3. ARCHITECTURE

Our Real-time Analytics Data Pipeline is a pipeline of loosely coupled stages. Each stage is functionally separate from its neighboring stage. Events are transported asynchronously across a pipeline of loosely coupled stages. This provides a better scaling model with higher reliability and scalability. Each stage can be built and operated independent of the neighboring stages and can adopt its own deployment and release cycles. The topology can be changed without a cold start.

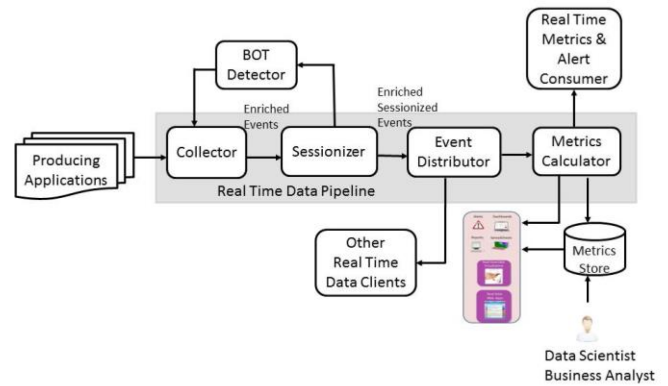

Figure 1. Real-time Data Pipeline

The pipeline extends all the way from producing services and applications in to our backend real time analytics infrastructure.

Event Streams carry user activity captured at source and published as events in to the pipeline. The events carry contextual information which is enriched and analyzed at various stages in the pipeline.

The events can be analyzed both in-flight and at rest. For in-flight analytics we flow the event streams into a stream processing infrastructure with Complex Event Processing (CEP) capability. This infrastructure is targeted for real-time use cases that require the output of analytics to become available to various consuming applications with a maximum latency of 2 seconds.

This infrastructure is targeted for real-time web analytics, personalization, bot detection, campaign monitoring, data quality measurement, mobile device error monitoring, alerting, correlation and many more such use cases.

Our approach is to treat the event stream like a database table. We apply SQL queries on live streams to extract summary data as events are moving. This technique is opposite of what one would do with batch processing where raw events are first collected in storage and then Map-Reduce jobs process the data at rest to produce summary data. We believe this will enable us to produce metrics much quicker with lot less resources. It also gives us the flexibility of deploying queries at runtime without code role out.

3.1 Complex Event Processing Framework

Our proposal uses Complex Event Processing (CEP) engines for a lot of our processing. We will supplement CEP engines with custom processors when CEP is not adequate for the processing or is too heavy.

Our data pipeline is a directed acyclic graph (DAG) of processing nodes as shown in Figure 2.

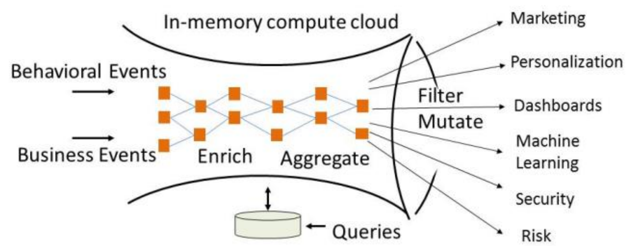

Figure 2. Processing pipeline graph

Each processing node hosts an instance of a CEP application which itself has a DAG of stages of its own. A collection of processing nodes in this graph performing the same operation forms a functional stage in the pipeline. The pipeline consumer's processing nodes form the leaf nodes of this DAG. The DAG keeps expanding as we add more and more consumers in the process changing the topology. We wanted a declarative way of stitching the pipeline so that we could make changes to the topology at run time.

Our processing logic involves processing event streams to compute metrics over tumbling and rolling windows, correlating streams, filtering, decorating and mutating streams, controlling the flow of events through the DAG, joining multiple streams and managing state.

The expectation was for the system to be very agile where processing logic could be written and deployed without a need for expert programmers. Most technical staff in our organization is usually familiar with SQL and hence we decided to support SQL like language as a declarative language for writing processing rules. The SQL statements are compiled to java byte code at run time so the real-time query processing performs very well. We also needed to extend SQL so the pipeline became visible to SQL thus enabling processing logic to control data flow in the pipeline.

We wanted our applications to be deployable in the cloud. The pipeline topology spreads across multiple data centers so we had complete disaster recovery.

Given our unique set of complex event processing requirement we decided to develop our own Spring[8] based distributed complex event processing infrastructure software called **Jetstream[2]**. It provides us a java complex event processing framework and tooling to build, deploy and manage complex event processing applications in our cloud environment. Tooling is also provided to deploy the apps in EC2 environment in docker containers.

Jetstream has the following capabilities:

1. Declaratively define processing logic in SQL
2. Hot deploy SQL without restarting applications
3. Annotation plugin framework to extend SQL functionality
4. Pipeline flow routing using SQL
5. Dynamically create stream affinity using SQL
6. Declarative pipeline stitching using Spring IOC enabling dynamic topology changes at runtime.
7. CEP capabilities through Esper [1] integration.
8. Clustering with elastic scaling
9. Cloud deployment
10. Pub Sub messaging with both push and pull models

All our real-time pipeline applications are built using Jetstream.

3.2 Messaging

3.2.1 Ingest Messaging

On a typical day we ingest up to hundreds of thousands of events/sec from more than 100,000 nodes. Average payload size varies between 1000 to 3000 bytes. Our producers operate in a polyglot environment. The event data is mostly unstructured with unpredictable patterns. We also have a requirement to keep the end to end latency in the system within 2 seconds driven by some of our targeted use cases.

Our primary choice is to ingest all our ingress event traffic via REST interfaces. Event payload is represented in either JSON or AVRO form and is batched at source with the batch size not exceeding 40 events. The events flow over periodically recycled persistent http connections. Data is compressed in-flight to optimize network bandwidth. The REST API provides us transactional semantics ensuring that the data made it to our collector stage. This also nicely solves our problem of integrating with our polyglot producer environment.

3.2.2 Push vs. Pull Messaging

For the messaging between distributed stages in our real-time pipeline we had to decide between a push vs. pull model.

The push messaging model offers very low latency and is less expensive as it has no persistent queues between distributed stages. However, its drawback is in dealing with slow consumers. There is a potential for losing messages if the consumer cannot keep up.

The pull model on the other hand has higher latencies and is more expensive as it involves persistent queues. It also causes amplification on the wire. However, this model is very suitable for slow consumers and very reliable.

3.2.3 Messaging Options

Many of the class of problems we are addressing demand very low latencies (less than 100 milliseconds) in the pipeline but at the same time demand very low steady state transport loss (less than 0.01%). In a cloud deployment the processing nodes can be spread across multiple data centers. Since a lot of our computation maintains aggregates in memory, we required our messaging tier to provide us a clustering capability in order to scale. Additionally, we wanted the messaging tier to support stream affinity to nodes in the consumer application cluster. Since our consumers could join and leave the pipeline at will we required a pub-sub messaging paradigm so our distributed clusters have loose coupling.

Jetstream offers a broker less in-memory very low latency pub-sub cluster messaging solution. It supports stream affinity through event schedulers that support consistent hashing. It also supports a random scheduler to distribute the load evenly among the cluster nodes. It provides us a capability to deploy our application cluster across datacenters. It has flow control semantics where by a consumer can signal to the producer to stop sending events to it causing traffic to be rebalanced across the cluster. It has the ability to detect a slow consumer and send advisories containing undelivered event. A listener can direct the undelivered event to a persistent queue to be replayed later.

Jetstream also supports guaranteed pub-sub messaging using Kafka which offers us at least once delivery semantics. We combine this with our broker less messaging to store and replay events that could not be delivered to slow consumers or when we have processing exceptions. We also optionally use this messaging to forward events to pipeline consumers when the requirement is skewed towards reliable messaging against low latency.

We picked a hybrid approach to use in-memory cluster messaging with persistent queues in exception path. This decision was mainly driven by cost and latency considerations. The persistent queue approach causes amplification on the network for the events flowing through our distributed multistage pipeline and adds a latency of at least 300 milliseconds between each distributed stage. The four stages combined would have added latency to the tune of 1.2 seconds. The latency would have been lot higher if we factored in our cross DC deployment requirement. This was not acceptable for many of our targeted use cases.

Our approach of combining in-memory messaging with overflow to persistent queue for exception path has worked very well for us. We are able to achieve less than one hundred millisecond end to end latency in the real-time data pipeline with steady state loss of less than 0.01%.

4. REAL-TIME PIPELINE

Our Real-time Analytics Pipeline is made up of the following four distinct application stages - Collector, Sessionizer, Distributor and Metrics Calculator. Each stage is an application cluster.

4.1 Collector

The Collector application is a Jetstream CEP Application deployed as a cluster in our cloud spanning multiple DCs. It is the first stage of our real-time pipeline. It interfaces with the producers of events on one side and then streams the events to the downstream Sessionizer stage. This stage is stateless. It exposes a REST interface to ingest events from producers. As events arrive they are validated for quality issues. Validated data is next passed through the CEP engine. The CEP engine filters out BOT events by looking up BOT signatures in a BOT signature cache.

4.1.1 Geo and Device Classification Enrichment

Bot filtered events are enriched with geo location information. A lookup is performed on a hosted in-memory geo-location database using the IP address in the event to find the geo location information like city, country, continent, region and line speed. One of the fields in the event contains an IP address. The geo-location database is populated by processing periodic feeds from a geo location feed vendor. This data is compressed into a bucketized binary tree making it extremely efficient for spatial searches. We are able to lookup geo information in less than 150 micro seconds.

The collector also hosts a device classifier which parses the user agent string to determine the device type, OS version and other device classification information.

The Agent String in the tracking event will be processed by the device classifier to identify the device associated with the user agent. The event will be decorated with the device tags for analysis downstream.

4.2 Bot Detector

In our environment, BOT signatures are looked up in our application tier way up in the producing applications. Signature of self-declared bots and those detected during offline processing is uploaded in a cache that is looked up. However BOTs that do not fall in this category will typically pass through to the real-time pipeline. These BOTs need to be detected as early as possible in the real- time pipeline so that we can filter the events associated with these BOTs before they start consuming valuable resources.

BOTs exhibit a certain pattern for accessing our site. Our concern is mostly with detecting BOTs that can consume a lot of our site resources, primarily compute, network and backend resources. Such BOTs can be detected by observing the rates at which the BOTs are accessing the site using specific signatures. Our approach is to use probabilistic frequency estimation techniques measured over rolling windows of time. In our opinion CEP, engines are best at detecting these patterns – it’s like finding a needle in a haystack.

As the engine detects BOT signatures it will update the BOT signature cache. This is looked up by the Collector to enforce BOT filtering.

4.3 Sessionizer

The Sessionizer is the second stage of the Real Time Pipeline. Its primary function is to support tenancy based sessionization.

Sessionization is a process of temporal grouping of events containing a specific identifier referred to as session duration. Each window starts when an event is first detected with the unique identifier. The window terminates when no events have arrived with that specific identifier for the specified duration referred to as session duration.

4.3.1 Session Meta Data, Counters and State

The data extracted from the events flowing into the session are stored in the session record in the form of session Meta data. Some examples of session meta data are SessionId, Page id, Geo-location (city, region, country, continent, longitude, latitude, and ISP, Browser, OS and Device type.

As events arrive we will maintain a count of the occurrence of user defined fields in those events or count the events. These counters are maintained in our session store. We also have capability to maintain state per session. We have the capability to set and reset state as we process events. All the processing logic is written in SQL.

4.3.2 Session Store

The session records are stored in a local off-heap cache which is backed up in a backing store. We had a requirement to be able to set the TTL on a cache entry and receive notifications with good precision when the entry expired. Since we did not find a COTS solution which could deliver this feature we developed a special off-heap cache with a runner so we can monitor when a cache entry expires and send notifications. Since the cache entries can be lost on node failure we store the entries in the off-heap cache also in an external backing store.

4.3.3 Session Backing Store

We used the following criteria for selecting the Session backing store:

a. Support local read, writes & deletes
b. Support both local and cross data center replication
c. Support for eventual consistency
d. Manage lifecycle of store entries (TTL support)
e. Support writes to read ratio of 10:4
f. Scale to 1M read &writes per second.
g. Scale well with variable size payloads from 200 bytes to 50000 bytes
h. Preferably deployable in the Cloud
i. Create secondary indexes for lookup of a range of keys inserted from a given client node

The session data is stored in off-heap cache and a backing store. We provide a storage abstraction for interfacing with different backing stores.

Our use case demands extremely high write and delete workloads. Any disk based solution will require compaction to handle deletes. This will require very large clusters and very expensive storage solutions. For our workloads a completely in-memory store with replication will scale best. We evaluated Cassandra[3], Couch base and a home grown store. We found both Cassandra and Couch base are disk based solutions and compaction becomes a bottleneck. Our choice was to go with a home grown solution which operates completely in-memory and provides cross DC replication with a fair degree of consistency.

4.3.4 SQL extensions

Jetstream provides an annotation plugin framework which enables users to write their own annotations to extend Esper's SQL like language. We have exploited this feature in Jetstream and developed special annotation to augment SQL. This allows us to write statements in SQL to perform the following operations:

a. Create sessions for a tenant specifying session duration and session identifier
b. Store meta data in session
c. Maintain counters in Session
d. Store & Manipulate state in session

For example, the following SQL is used to create a session and define the identifier and session duration.

```
@Session("WebSession")
       select si as _pk_, _ct as
_timestamp_, 30 as _duration_
       from RawEvent(si is not null
and _ct is not null);
```

To update a counter named "pageviews" in the session we use the following SQL

```
@UpdateCounter("pageviews")
      select * from
RawEvent(pageGroup = 'HomePage');
```

4.4 Event Distributor

This is the third stage in the pipeline and its primary function is to create custom views for pipeline subscribers. The views are created by mutating, filtering and routing sessionized streams. Pipeline subscribers use a pub-sub interface to subscribe to the events sourced by the distributor. The subscription is authorized by an authorization system enforcing that a subscriber's view is populated with data that the subscriber is authorized to see. Subscribers can join and leave the pipeline at will. When a subscriber joins the

pipeline their view becomes active. They start receiving events from the time the view became active.

4.4.1 Event filtering, Mutation and Routing

The rules for mutation, filtering and routing are all written in SQL and can be changed at run time. A sample SQL for mutation, filtering and routing is shown below.

```
 insert into PSTREAM select D1, D2, D3, D4  from RawEvent where D1 = 2045573 or D2 = 2047936 or D3 = 2051457 or D4 = 2053742;

 @PublishOn(topics="Trkng.Aconsumer/pEvent")
 @OutputTo("OutboundMessageChannel")
  @ClusterAffinityTag(column = D1)
 select *  FROM PSTREAM;
```

All the mutation and filtering is done in SQL syntax. Jetstream provides us annotation extensions to SQL which gives us visibility to the pipeline. We use these annotations to specify the route the event takes in the data pipeline once the select statement executes.

4.5 Metrics calculator for Multi-dimensional OLAP

Metrics Calculator is a real-time metrics computation engine which computes user defined metrics over various dimensions and produces time series data. It provides a SQL interface for users to submit SQL queries to harvest metrics by grouping multiple dimensions over tumbling windows of time. The metric event, an output produced by the engine, can be routed to one or more destinations - all controlled through SQL. The destination could be a time series data base or a visualization widget connected through a web socket interface or a consumer that needs to be alerted on a threshold crossing of a metric.

4.5.1 Harvesting Metrics

Events can be scheduled into the Metrics Calculator application cluster using either random scheduling or affinity based scheduling. For affinity based scheduling, an affinity is created between the event stream and one of the metrics calculator's processing nodes. An affinity key is composed using one or more dimensions in the event. The affinity key binds the event stream to a processing node in the cluster. This guarantees that events with same affinity key always land on the same processing node in the cluster, enabling us to maintain aggregates in memory.

The metrics are harvested from live streams over short tumbling windows (ten seconds). When the window rolls it produces a metric event for each of the unique dimension grouping.

A sample SQL query for harvesting metrics is shown below.

```
create context MCContext start @now
end pattern [timer:interval(10) or
EsperEndEvent];

context MCContext
insert into aggregate1
   select count(*) as count, D1, D2,
D3, _timestamp as tag_time, 'M1' as
metricName from RawEvent(D1 is not
null and D2 is not null and D3 is not
null) group by D1, D2, D3 output
snapshot when terminated;
```

4.5.2 Aggregating across the Metrics Calculator cluster

The metric computation shown in section 6.1 is complete as long as the affinity key is composed using one of the dimensions in the dimension grouping for the metric. However, this computation is not complete in cases where events are scheduled using random scheduling or when the affinity key for affinity scheduling is composed from dimensions that are not part of the dimension grouping.

The expected result of the computation is for the engine to produce 1 metric event for a unique grouping of the dimensions. For scenarios discussed above for a single metric to be produced in the cluster, all the individual metric events produced in each of the cluster nodes has to be directed to the same cluster node in the cluster for a unique grouping of the dimensions. A stream affinity needs to be created by composing an affinity key using the dimensions constituting the grouping. This is accomplished with Jetstream SQL annotations as shown below.

```
@OutputTo("outboundMessageChannel")
@ClusterAffinityTag(dimension=@Create
Dimension(name="groupdimen",
dimensionspan="D1, metricName, D2,
D3, tag_time"))
@PublishOn(topics="Trkng.MC/clusterLe
velAggregate")
select * from aggregate1;
```

4.5.3 Creating rollups in time series database

We compute the cluster level aggregate for the metric over another thirty second window. When the window rolls, the aggregated metric is output as a metric event. The engine is now producing time series data.

Sample SQL for creating cluster level aggregate and metric event directed to a time series database is shown below.

```
create context CAContext start @now
end pattern [timer:interval(30) or
EsperEndEvent];

context CAContext
insert into clusteraggregate select
SUM(count) as count, D1, D2, D3,
tag_time, metricName from aggregate1
group by D1, D2, D3, tag_time,
metricName;

@OutputTo("timeseriesdatabase,
visualizer") select * from
clusteraggregate output snapshot when
terminated;
```

The time series data produced can be recorded in a time series database or drive visualization widgets in a real time dashboard.

4.5.4 Metric Store

Our time series data store requirements were to ingest at a very high rate (a few hundred thousand events/sec), create rollups over different time windows (min, hour, day), submit adhoc queries to create aggregates grouped by any random combination of dimensions, query scan ranges spanning multiple years, query latencies under a few seconds for most cases, support over a hundred concurrent queries without impacting ingest rate and query latencies, highly available and support the following aggregate functions like SUM, , AVG, COUNT, TOP N, DISTINCT COUNT, PERCENTILES.

We evaluated Open TSDB[4], Cassandra[3] and DRUID[7]. Open TSDB supports high ingest and can support some of the aggregate functions we wanted. However, it does not allow us to create rollups and it also does not support creation of new aggregates from existing aggregates. Cassandra can also support high ingest. We can create rollups for different time windows using the Cassandra counter column family. However Cassandra does not support GROUP BY and the aggregate functions we want. Our choice is DRUID as it supports all our requirements.

5. CONCLUDING REMARKS

In this paper we have described the data and processing model for a class of problems related to user behavior analytics in real time. We describe some of the design considerations for Pulsar. Pulsar has been in production in the eBay cloud for over a year. We process hundreds of thousands of events/sec with a steady state loss of less than 0.01%. Our pipeline end to end latency is less than a hundred milliseconds measured at the 95th percentile. We have successfully operated the pipeline over this time at 99.99% availability. Several teams within eBay have successfully built solutions leveraging our platform, solving problems like in-session personalization, advertising, internet marketing, billing, business monitoring and many more.

Although we focus mostly on user behavior analytics which is our primary use case, we envision Pulsar to be used for many other use cases that require real-time processing.

REFERENCES

[1] Esper - http://esper.codehaus.org/esper/documentation/documentation.html

[2] Jetstream - https://github.com/pulsarIO/jetstream/wiki

[3] Apache Cassandra - http://cassandra.apache.org

[4] Open TSDB - http://opentsdb.net/

[5] Apache Kafka - http://kafka.apache.org/

[6] Efficient Computation of Frequent and Top-k Elements in DataStreams - https://icmi.cs.ucsb.edu/research/tech_reports/reports/2005-23.pdf

[7] DRUID - http://static.druid.io/docs/druid.pdf
The 8 Requirements of Real-Time Stream Processing - http://cs.brown.edu/~ugur/8rulesSigRec.pdf

[8] Spring - https://spring.io/

[9] HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm - http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

[10] Computing extremely accurate quantiles using T-Digest - https://github.com/tdunning/t-digest/blob/master/docs/t-digest-paper/histo.pdf