Rakam: Distributed Analytics API

Burak Emre Kabakcı

May 30, 2014

Abstract

Today, most of the "big data" applications needs to compute data in real-time since the Internet develops quite fast and the users expect the get reactions from the applications simultaneously. This rule is valid for almost all types of applications. When a user interacts with a commercial website by looking a product, the website should be able to show her related products for increasing its conversion rates. For a CRM application, the users should be able to solve their problem using that application. And most the time, these actions need aggregation computations. The aim of our project is to provide a high-performance scalable computational engine that is flexible and can be adapted to any type of applications. The system collects the events with collection API and continuously processes them on the fly with using pre-aggregation rules submitted by Analysis API.

1 Introduction

Rakam includes a list of APIs: Collection API, Analysis API, Aggregation Rule API and Actor API. Collection API simply collects the data and submits it to pre-aggregation rule processor threads. The current implementation only supports RESTful HTTP Server. We mainly use Vert.x [1] for coordinating nodes and listening sockets for webservers. It's is a lightweight, high performance application platform for the JVM using asynchronous Netty [2] framework. The asynchronous nature of Netty provides us to create high-performance webservers using Java. In Aggregation Rule API the users can submit aggregation rules to the system via RESTful HTTP interface.

There are two types of aggregation rules: Metric aggregation rule and Time-series aggregation rule. Metric aggregation rule is simply a global data produced by the collection workers. However time-series aggregation rule creates new metric for each interval given by users. The strategy of aggregation rules specifies the order and addresses of processing the rules. The currenct implementation has four different strategy: *REAL_TIME_BATCH_CONCURRENT*, *REAL_TIME*, *REAL_TIME_AFTER_BATCH* and *BATCH*. Since the batch processing can take much more time to evaluate, it's important for users to be able to specify to strategy of aggregation rules.

The aggregations types are: *COUNT*, *COUNT_X*, *COUNT_UNIQUE_X*, *SUM_X*, *MINIMUM_X*, *MAXIMUM_X*, *AVERAGE_X* and *SELECT_UNIQUE_X*.

COUNT simply increments a counter related to specified type of event. It's useful for tracking general information about data aggregated. *COUNT_X*,

COUNT_UNIQUE_X, *SUM_X*, *MINIMUM_X*, *MAXIMUM_X*, *AVERAGE_X* aggregation types uses specified field to create a metric. It generates a simple long value as a result. *COUNT_UNIQUE_X* and SELECT_UNIQUE_X keep a set for counting unique things on the fly. All aggregation types supports group by feature. It groups items by extracting key from group by attribute. Also Rakam supports filters of events. When aggregating the data it checks filters and if the event doesn't conform, it simply ignores the event.

These three types of fields (select, group by) inherits from an interface called FieldScript and can be a MVEL [3] script or an attribute. Scripting provides us flexibility of grouping and selection. Users able to apply functions on fields or create tuples of field values using MVEL scripting feature. Filters also support scripting, FieldScript expects a Boolean value and if the script produces a false it ignores the event. When new aggregated rule submitted by user, the node serialize aggregation rule and send it to all nodes in the cluster. Since the aggregation rules are small objects, they don't occupy much space in Java heap. However replicating data among the cluster eliminates the network requests between collection workers and distributed cache layers.

Analysis API is the query interface for aggregation rules. Users are able to query the aggregated results using the unique id for aggregation rules. In our tests, the response time is under 500ms under heavy load (3000 event/s). The last API in our system is Actor API. There is a special field called _user in collection API. This attribute directly maps to another table called actor in database layer. Actor API is quite useful for real-time analytics because it keeps track of statistics about event actors. It can also be used for A/B testing and CRM applications that take advantages of Rakam. Since the batch requests for this special attribute requires JOIN operation and collection workers must store actor id and property mappings, it comes with an extra overhead so we made it optional. When a user sends a request to user API the system ignores events, which don't have _user attribute. The function of this API is to allow querying actors via an interface.

2 Discussion

2.1 Problems / Solutions

We explained features of Rakam in previous section. Now we will explain the internal mechanism of the system and the problems we faced and solutions we came up.

2.1.1 Replicated Aggregation Rule Map

As we said earlier, our main aim is to provide a high-performance system but it must also be consistent and reliable. Our first problem was about synchronizing aggregation rules across the cluster. Since it will be needed in collection workers for processing events, we should keep a local copy of data to avoid extra network request to cache or database because it dramatically increases average processing time for events. We created a ConcurrentHashMap in Java since it provides us faster read time. When a new aggregation rule submitted by user, the load balancer selects a webserver in a node in cluster. The webserver sends

this request to a worker thread for processing since we shouldn't block the webserver thread to be able to get higher requests per second. The worker thread deserialize the message and if it's a add request, it save the aggregation rule to database and local copy of aggregation rules. Also it sends aggregation rules to the all nodes in the cluster using event bus. Event bus is a mechanism provided by Vert.x. Each node has a unique id and connected the each other via a event bus mechanism. Normally the event bus in Vert.x only supports the verticles in same node but using Hazelcast [10] we handled the cluster management issue. Hazelcast allows us the use its internal cluster management feature externally. We starts a Hazelcast instance from a node with shared configuration that specifies the multicast and TCP behaviors and when another node create another Hazelcast instance with using same configurations the nodes automatically connects to each other and we can create a TCP connection which acts like an event bus across all nodes in cluster. Using this event bus, we send the aggregation rule to the all nodes by publishing a message, when the message arrives a node, the node checks the timestamp value of message and if it's higher than the last write timestamp, we process the message and save the aggregation rule to local ConcurrentHashMap. Otherwise we ignore the message. This process is same for add, update and delete requests because the messages are unordered and may arrive multiple times to the nodes. We send the data if we can't get a response for a certain interval. Storm [4], a distributed RPC for continuous stream processing engine, uses a similar mechanism. [12] When the new aggregation rule is replicated to the nodes, the collection workers become aware of the new aggregation rule and use it for continuous aggregation processing. In our tests this technique dramatically increased number of requests processing per second from 6102 requests/s to 10453 requests/s.

2.1.2 Periodic Local Collectors

The other problem was related to cache mechanism of the system. When we started developing the application, we were using two data storage layers: cache layer and database layer. Collection workers were doing modification on cache layer and in another thread, we were running a periodic task that runs each seconds and collects the data from cache layer and saves it to the database layer. A similar technique is proposed in a paper related to continuous analytics. [13] And in analysis API, the data were coming from only database layer. This algorithm was good but we needed a better one because in higher throughput, the transmission of data between worker thread and cache layer were causing too much overhead and reduces the performance of worker thread. Instead, we developed a new cache layer and finally we had end up three data storage layers. The new cache layer is the local cache adapter that keeps the data in memory. Since it's not distributed it's quite fast to modify the data in this cache layer. Similar to our first algorithm, in each second a periodic task is executed by the system that moves all data in first cache layer (local in-memory cache) into the second storage layer (distributed in-memory cache) and also moves some of the infrequent data in second cache layer to database layer (file-system storage). This three storages mechanism increased our collection workers performance by x1.5.

However even if this problem increases workers performance, it causes data loss if a node faults before saving data in first layer cache to second layer cache.

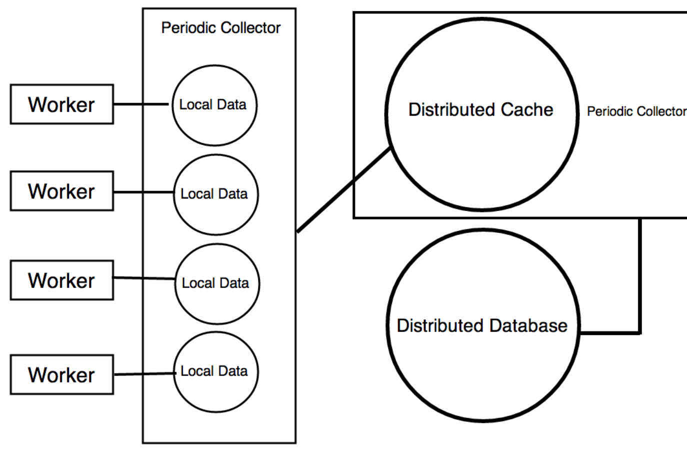

Figure 1: Graph representation of storage layers

Since the second cache layer is distributed and the replication factor is at least two, the faulty nodes are not problem until the half of the nodes are down but the local cache is a real problem and make the system unreliable. Luckily, we store the event data to database layer synchronously in collection workers because we can revive the data from raw event table into our KeyValueStorage that keeps the aggregation results. Since our current database storage is Cassandra and it's write performance is quite good when we use eventually consistent model, storing raw event data in a blob column in Cassandra doesn't take more than 3ms in our tests.

When a node is down, the event-bus informs each nodes but it takes at least a few seconds since the event-bus assumes the connection is slow when a node responds late or doesn't response. It waits for the result until timeout. And it tries to create a new connection between that node and after a few tries; if it doesn't get any data from that node it assumes the node is down. To be able to find out the timestamp of periodic task's last run, we developed a check-in mechanism. In each run, periodic tasks check-in to the database layers with a timestamp value. When a node is down, the nodes get informed and one of the nodes look-up the timestamp of the node's last run and send a new asynchronous batch request to the database layer with the node id. The database layer automatically processes all aggregation rules and updates the corresponding aggregated data segments. There are similar academic works on the issue. Spark Streaming [5], a distributed stream-processing engine, uses a similar mechanism. They divide the data into small blocks and perform batch processing in these small blocks. RAMCloud [9] focuses on this issue and propose a similar technique as a solution. In our mechanism, the aggregate the data on the fly and move it between storage layers. In the end, the performance of most important part of the project dramatically increases and we still have a consistent and reliable system.

2.2 Modularity

The most parts of the project are modular and can be replaced to another solutions when needed. For example the current implementation for database storage support Cassandra. Cassandra [6] is distributed key-value storage and provides us consistent and reliable data storage. However it doesn't support any form of ad-host queries. The batch process part of the Cassandra uses Hadoop API with Spark [7] batch processing and in our tests, when the data grows to more than 10 million events, a generic batch processes which includes group by takes more than ~15 minutes. However we can replace Cassandra with Elasticsearch, which is a distributed search engine and the batch process should take much more less time. Also we can use RDBMS with partitioning if we need low memory footprint.

Also we use Google Guice [8] for dependency injection. Rakam allows event mapping and filtering with external mappers. When we start the program, it checks the configuration file we provided and caches mapper and filter plugins in that configuration file. Collection workers use these plugins for mapping or filtering event data. For example if we're developing a website analytics software, we may need the location information of the visitors. Since the only information about location is IP, we need to resolve the IP into country, city, latitude and longitude information when saving and processing event data.

2.3 Query optimization

We aggressively check new aggregation rules when adding to the system. For example if the aggregation type is AVERAGE_X, we split the aggregation rule into SUM_X and COUNT_X and save it as its new form. Therefore, if COUNT_X is already added to the system, we use it for calculation of AVERAGE_X aggregation rule. Also SELECT_UNIQUE_X and COUNT_UNIQUE_X share same data and when one of them is added to the system, users are able to query the other one.

2.4 Re-computation of lost data

When a node is down, the local data is lost so we need to calculate the aggregated data again. Once we detect a node failure, we send a batch request of that interval in order to recover the lost state. However when data grows, batch requests take too much time and it affects the consistency of the program. Since the data is updated at a specific interval depending on the interval of periodic scheduler we need to recover the state in that specific period. Rakam doesn't includes an internal database that uses file-system, we uses database adapter for storing events so we need to recover the state from database layer. The current implementation of database adapter uses Cassandra [10] and batch requests use Spark for Hadoop processing. Since Hadoop processes take too much time we store the event data with it's timestamp, the node id of the collection worker and thread id. Since these three combinations are completely independent and node ids' are unique we can assure that there won't be any collision. We use Java long primitive for timestamp representation in milliseconds and it occupies 8 bytes. The node id and thread ids are using short primitive and it occupies 2 bytes. The previous implementation were using TimeUUID and it occupies 16

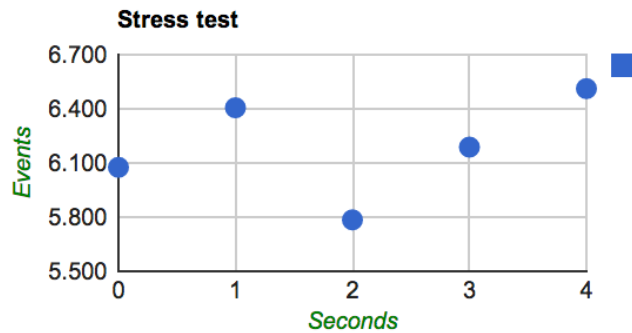

Figure 2: Rakam stress test results

bytes but with this method, we use 12 bytes and the new id generation system allows us to identify node is for state recovery. Therefore we perform query for fail interval and node id and process event in a worker in the cluster instead of performing map-reduce request in CassandraFS. It also guarantees that there is no collision between event ids. Twitter uses a similar method for distributed id generation and open sourced their project Snowflake [11].

3 Results

We have proposed a new term aggregation rule and used it for continuous stream processing. Rakam uses several distributed algorithms for continuous computation effectively. It uses three storage layers for storing data and continuously moves data between storage layers depending on their tradeoffs. Rakam aims to store aggregation rule efficiently and splits it into multiple aggregation rules in order to share data between aggregation rules. It also aims to recover lost state in a short time efficiently. A Rakam node can collect events up to 50K/second with RESTful API depending on the count of aggregation rules in the system.

4 Future Work

Since Rakam is quite generic solution for analytics, it's useful for almost all form of Analytics applications. For analytics part, our first aim is to make Rakam stable. Then, we're plannning to add more features to analytis API. For example, time-series aggregation rules should be combinative. When user adds an aggregation with one hour interval, she can also be able to query multiple of one hour. Implementation is eacy for aggregation rules which requires counters because summing long primitives is not computationaly expensive in Java. However, for aggregation rules which requires set, since sets can grow very large it can take too much time. At this point, we're planning to use HyperLogLog [14] algorithm for cardinality estimator. Users should be able to speficy the error rates and depending on these rates, we will change the underlying map size in HyperLogLog data structure.

References

[1] Vert.x *http://vertx.io/*

[2] Netty *http://netty.io/*

[3] MVEL *http://mvel.codehaus.org/*

[4] Storm *http://storm.incubator.apache.org/*

[5] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing, December 14, 2012

[6] Cassandra *http://cassandra.apache.org/*

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

[8] Google Guice *https://code.google.com/p/google-guice/*

[9] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In SOSP, 2011.

[10] Hazelcast *http://www.hazelcast.com/*

[11] Snowflake *https://github.com/twitter/snowflake/*

[12] N. Marz. Trident: a high-level abstraction for realtime computation. *http://engineering.twitter.com/2012/08/trident-high-level-abstraction-for.html*

[13] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. CIDR, 2009.

[14] Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm 2007