Query-able Kafka: An agile data analytics pipeline for mobile wireless networks

Eric Falk
University of Luxembourg
eric.falk@uni.lu

Vijay K. Gurbani
Bell Laboratories, Nokia Networks
vijay.gurbani@nokia-bell-labs.com

Radu State
University of Luxembourg
radu.state@uni.lu

ABSTRACT

Due to their promise of delivering real-time network insights, today's streaming analytics platforms are increasingly being used in the communications networks where the impact of the insights go beyond sentiment and trend analysis to include real-time detection of security attacks and prediction of network state (i.e., is the network transitioning towards an outage). Current streaming analytics platforms operate under the assumption that arriving traffic is to the order of kilobytes produced at very high frequencies. However, communications networks, especially the telecommunication networks, challenge this assumption because some of the arriving traffic in these networks is to the order of gigabytes, but produced at medium to low velocities. Furthermore, these large datasets may need to be ingested in their entirety to render network insights in real-time. Our interest is to subject today's streaming analytics platforms — constructed from state-of-the art software components (Kafka, Spark, HDFS, ElasticSearch) — to traffic densities observed in such communications networks. We find that filtering on such large datasets is best done in a common upstream point instead of being pushed to, and repeated, in downstream components. To demonstrate the advantages of such an approach, we modify Apache Kafka to perform limited *native* data transformation and filtering, relieving the downstream Spark application from doing this. Our approach outperforms four prevalent analytics pipeline architectures with negligible overhead compared to standard Kafka. (Our modifications to Apache Kafka are publicly available at https://github.com/Esquive/queryable-kafka.git)

1. INTRODUCTION

Streaming analytics platforms, generally composed of tiered architectures as epitomized by the Lambda Architecture [26], promise to deliver (near) real-time decisions on large, continuous data streams. Such streams arise naturally in communications networks, especially large networks like Twitter, Facebook, and packet-based public telecommunication system like the existing 4G networks. Architectures inspired by the Lambda Architecture are organized in three layers: batch, streaming and serving layer. Incoming data streams are distributed simultaneously to the batch and streaming layers, with the assumption that there is a sizeable time lag between fitting the arriving data to models in the streaming and batch layers. The streaming layer operates on a data only once, in the order that it arrives, to deliver results in near real time. The batch layer has the luxury of iterating over the entire dataset to produce results, at the cost of time. The batch and streaming layers forward computational output to the serving layer, which aggregates and merges the results for presentation and allows ad-hoc queries.

While the streaming analytics architectures used today suffice for large networks like Twitter and Facebook, this paper shows that they are not designed to handle the volume of packet-based telecommunications systems like 4G or its successor, the 5G network [6]. Global mobile devices and connections in 2015 grew to 7.9 billion; every server, mobile device (phone, tablet, vehicles), and network element in 4G generates a steady stream of data that needs to be sifted through and processed with real-time requirements to predict anomalies in the network state or foreshadow attacks on the network. To characterize the inability of existing streaming analytics architectures to handle the volumes observed in the telecommunication traffic, we briefly describe an important server in 4G and contextualize it's traffic volume to that of other large networks like Twitter and Facebook.

In the 4G network, a server called the Mobility Management Entity (MME) is a key control node for the access (radio) network. It tracks the mobiles (or more generally called a UE, User Equipment) as they move across cell boundaries, pages the UEs when messages arrive for them, authenticates the UE with the network, and is responsible for assigning and maintaining the set of network parameters that define specific treatment accorded to packets from that UE. In short, the MME is involved in all decisions that allow the UE to interface with the network. The MME produces a logfile record of every interaction between the UE and other network elements; the logfile record contains over 230 fields, and each logfile record is about 850-1000+ bytes long. The logfile is maintained in a CSV format in UTF-8 encoding.

MMEs are configured for redundancy and reliability and deployed in a pool to serve a metropolitan area. In a large metropolitan area of 10 million people, under the conservative estimate that 20% of the UEs are interacting with the network (establishing Internet sessions, moving around,

Proceedings of the VLDB Endowment, Vol. 10, No. 12

making phone calls, etc.) the MME pool serving this area will generate data at a rate of about 40 MB/s, or 3.5 TB/-day. (In a large metro area, a mean of 50,000 records/s will be generated by the MME with each record weighing in at 850 bytes for a rate of 42.5 MB/s.) By contrast, the WhatsApp instant messaging service, with a mean of 347,222 messages/s [1], at an average message size of 66 bytes (with metadata), generates about 22.9 MB/s[1]. The Twitter network generates 11.4 MB/s[2], and finally, Zhao et al. [41] state that Facebook generates log data at the rate of 289.35 MB/s. To ensure our comparison is uniform across the MME logfile data and other networks, we restrict our analysis to textual data from the Twitter, Facebook, and WhatsApp networks. Our metric does not include media attachments — video or pictures — of any kind.

Table 1: Ranking of message traffic in terms of MB/sec. (Only considering textual data)

Rank	Use Case	Traffic in MB/sec.
1.	Facebook	289.35
2.	Pool of 5 active MMEs	200.00
3.	Single MME	40.00
4.	WhatsApp	22.90
5.	Twitter	11.40

It is clear from the above table that datasets in telecommunication networks are in the upper end of the data volumes generated by other networks. The provided measures quantify the data generated by a MME of a 4G network entity, and it is expected that the 5G network with its support for Internet of Things, programmable networks, autonomic computing, and further thrusts into vehicular ad hoc networks will push the need for even larger datasets for prediction and analytics.

1.1 Problem statement and contributions

A detailed description of our application in §3, here we motivate the problem statement and contributions.

The MME collects the logfiles in increments of a minute, i.e., each minute, all the data is written out to the log file. While it is possible to obtain the data on finer timescales, the 1-minute batching mode suffices for our particular application on predicting a network outage because the logfile contained enough observations to allow the data to fit our statistical models. At the rate of 40 MB/s, the MME logfile can grow to 2.4 GB/min. Ingesting such a large file in a real-time data pipeline imposes latency penalties driven by the fact that the entire 2.4 GB file must be processed before the next batch arrives. Our models create a distribution from the arriving data, requiring access to the entire minute. Sampling was eschewed because it may not catch aberrant datapoints, which may be outliers of interest. The models are discussed in Gurbani et al. [17] and Falk et al. [14]. Here, we don't discuss the models themselves but present the systemic challenges faced and mitigated while constructing the agile data analytics pipeline. Without such an agile streaming analytics pipeline we had an unstable system that could not render timely predictive decisions.

[1]Numbers are based on Vergara et. al [37]. The average message size is 26 bytes plus 40 bytes of metadata for a total of 66 bytes.

[2]According to Twitter blogs [28], the network observes a mean of 5,700 tweets/s at about 2 KB/tweet, including metadata. This results in 11.4 MB/s.

As mentioned above, the entire 2.4 GB file must be processed before the next batch arrives. Furthermore, the nature of the streaming analytics data platforms in use today (i.e., the Lambda Architecture) is to perform data transformation in the speed or batch layers; consequently, if there are multiple speed layers (as there are in our application), the large dataset has to be transmitted to the order of the number of speed layers. Each speed layer will only need a subset of the columnstores in the dataset, however, due to the paucity of transformation tools at the ingestion point, the large dataset must be transmitted in its entirety to each layer. While streaming analytics platforms are horizontally scalable, their distributed data storages cannot deliver without some latency, which keeps growing as the size of datasets increases. Mining such big data within computable time bounds is an active area of research [15].

This work contributes to data management research through the following three contributions:

- We construct and evaluate a telecommunications-centric real-time streaming data pipeline using state-of-the-art open source software components (Apache Hadoop [3], Spark [39] and Kafka [23]). During our evaluation, we subject the analytics pipeline to workloads not typically seen in the canonical use of such platforms in global networks like Facebook and Twitter. Our findings are: (A) traditional analytics platforms are unable to handle the volume of data in the workloads typical of a telecommunication network (large datasets produced at medium to low velocities). The primary reason for this is the inability of the speed layer to deal with extreme message sizes (2.4 GB/min). Aggregating, summarizing or projecting relevant columns at the source (MME) is not appropriate in our application due to stringent constraints that monitor production-grade MME resources to keep them under engineered thresholds (discussed in more detail in §4.4). (B) It became increasingly clear that the batch layer was not needed in our application. The speed layers expeditiously compare arriving target distributions to known distributions that were created a-priori. Our application is not driven by ad-hoc queries — What is trending now? How many queries arriving from the southern hemisphere? — as much as it is by instantaneous and temporal decisions — Is a network outage in progress? Is the network slice overloaded?

- To overcome limitations of prevalent streaming analytics platforms, we propose to move extract-transform-load (ETL) tasks further upstream in the processing pipeline. This alleviates our finding (A) above and allows the analytics platform to process typical workloads in a telecommunication network. A novel, simple on-demand query mechanism was implemented on top of Apache Kafka, the ingestion point in our framework. This mechanism is located close to persistent storage to leverage capabilities of disk backed linear search, alleviating each speed layer to deal with the ETL overhead. We call our approach the *query-able Kafka* solution.

- We present an extensive evaluations of four streaming analytics frameworks and compare these to our proposed framework implementing our novel on-demand

query mechanism. We evaluate the results of these architectures on a cluster of 8 hosts as well as on Amazon AWS cluster. For each cluster, we measure three metrics: CPU consumption of the hosts in the streaming analytics platform, memory consumption and time-to-completion (TTC), defined as the time difference between the message emission from the MME to the insertion of model output from the speed layer into the serving layer.

The rest of the paper is organized as follows: §2 overviews related work, §3 outlines our target application used in the streaming analytics pipeline, and §4 outlines four prevalent streaming analytics architectures and introduces our queryable Kafka solution. §5 evaluates our solution against prevalent architectures and discusses the advantages of our approach; we conclude in §6.

2. RELATED WORK

By the late 90's databases for telecommunications use cases were an active area of research [19]. To maintain quality of service and enable fault recovery, massive amounts of log data generated by networking was analyzed close to real-time. Two notable works emerging from the telco domain, addressing this challenges in the pre-Hadoop era are: Gigascope [11] from AT&T research labs, and Datablitz from Bell Laboratories [7]. Gigascope is a data store built to analyze streaming network data to facilitate real-time reactiveness. Datablitz is a storage manager empowering highly concurrent access to in-memory databases. Both solutions were designed to serve time critical applications in the context of massive streaming data.

Hadoop [3] and MapReduce [12] revolutionized large scale data analytics, and for a long time appeared as the all-purpose answers to any big data assignment. Hadoop cannot be used in the cases where Gigascope and Datablitz are deployed because the out-of-the-box Hadoop is not suited for real-time needs; nonetheless, Facebook and Twitter have attempted to use it as a real-time processing platform [10][27]. Further approaches have been made to port MapReduce operations on data streams [24] and proper stream processing frameworks have emerged [35][39][2].

When it comes to big data, the missing capacity if compared to traditional RDBM systems, is the time to availability once data is ingested. For a RDBMS, data is instantaneously available whereas with big data stores a batch task usually has to run in order to extract the information and make the data eligible for ad-hoc queries. Although being horizontally scalable, distributed data storages cannot deliver without noticeable latency. Moreover this latency keeps increasing in front of the sheer amount of data to process. The issue is not solved yet [15], among the proposed solutions two protrude: the commonly called Lambda architecture [26] and the Kappa architecture [22]. The Lambda architecture capitalizes on the facts that models issued by long term batch processes are strongly consistent because the full dataset can be examined, whereas models from stream processors are fast to generate but only eventually consistent since only small data portions are inspected [26]. The architecture consists of three layers: the batch, speed and the query layer. The batch layer is typically build on top of Hadoop, and the speed layer employs distributed stream processors as Apache Storm [35], Spark [39] or Flink [2]. The Lambda architecture assumes that eventually consistent models, from the speed layer, are tolerable to bridge the time required for two consecutive batch layer runs to execute. The batch layer is the entity generating the consistent models, replacing the stream layer model once completed. Examples of implemented Lambda architectures are in [26][38]. Analytics in the telecommunication domain is more concerned with the currently arriving dataset instead of the archived ones; it is crucial to employ the incoming stream data to contextualize the network state. Leveraging a full fledged Hadoop cluster as a batch layer is of minor importance for real-time analytics of 4G MME data. An alternative to the Lambda architecture is the Kappa architecture [22]. The Kappa architecture employs the distributed log aggregator and message broker Kafka [23] as persistent store, instead of a relational database or a distributed storage. Data is appended to an immutable logfile, from where it is streamed rapidly to consumers for computational tasks. The totality of the data stored in the Kafka log can be re-consumed at will, while replicated stream processors are synchronized to ensure an uninterrupted succession of consistent models. Works such as [8] and [25] investigate big data architectures for use with machine learning and network security monitoring, respectively. In the same manner a recent showcase of the Kappa architecture deployed for a telecommunication use case is given by [33]. In the Kappa architecture implemented in [33], the stream processor consumes raw data from Kafka, transforms it and writes it back to Kafka for later consumption by the model evaluation job. We demonstrate in this paper that this pattern is not efficient with large MME logfiles.

Literature on streaming data transformation is found in the data-warehousing community. For instance Gobblin [31] from LinkedIn unifies data ingestion into Hadoop's HDFS through an extensible interface and in-memory ETL capabilitites. Karakasidis et al. [21] investigate dedicated ETL queues as an alternative to bulk ETL on the persistent store. Their experiments show that isolated ETL tasks only add minimal overhead but enable superior performance and data flow orchestration. They also establish a taxonomy of transformations, categorizing operations in 3 classes:

- Filters: incoming tuples have to satisfy a condition in order to be forwarded. In relational algebra this corresponds to select [16]: $\sigma_C(R)$ operator. C is the condition, and R the relation, in other words: the data unit the selection is applied to.

- Transformers: the data structure of the results is alternated, in analogy to a projection: $\pi_{A_1,A_2,...,A_n}(R)$. $A_1, A_2, ..., A_n$ are the attributes to be contained in the results, and R the relation. Other transformers in the sense of [21] are per row aggregations, based on attributes of the relation.

- Binary operators: multiple sources, streams in this case, are combined to a single result. The equivalent in relational algebra terminology is the join: $R_1 \bowtie R_2$, where R_1 and R_2 are two relations joined to a single resultset.

Babu et al. [5] survey continuous queries over data streams are surveyed in the context of *STREAM*, a database for streaming data. A constantly updated view on the query

results is kept, which implies deletion of entries that become obsolete. Similar to the query layer of a Lambda architecture, an up-to-date view on the data is maintained in a summary table. With respect to our target application, the continuous queries could be applied to each MME logfile as it transits the data through the pipeline. One consideration in that direction was Apache NiFi [4], which supports transformations on CSV files through regular expressions [30]. Treating CSV files without proper semantics is a risky undertaking because not every boundary case can be foreseen. Furthermore, in it's current state, Apache NiFi is not appropriate for bulk processing [29], which is what we need in our on-demand queryable Kafka solution. Finally, In *JetStream* [32], a system designed for near real-time data analysis across wide areas, employs continuous queries for data aggregation at the locations of the data source. The condensed data is transmitted to a central point for the final analysis. However, *JetStream* exceeds the time constraints imposed by our underlying task even when dealing with smaller datasets that we use in our work.

3. TARGET APPLICATION

Our analytics platform supports a predictive model of the network state, i.e., using the logfile records produced by the MME (c.f. §1), the application utilizes the analytics pipeline to predict the state of the network in near real-time. This near real-time information provided the network operator an authoritative insight of the network, which would not otherwise be possible. Network operators monitor these log messages on a best-effort basis as real-time monitoring is not always possible, leading operators to be in a reactive mode when an outage occurs.

The MME produces a logfile record of every interaction between the UE and other network elements, each such interaction is called a *procedure* and the record contains a unique *ProcedureID* for each procedure; there are about 70 procedures defined. Besides the *ProcedureID*, the logfile record contains over 230 fields, and each logfile record is about 850-1000+ bytes long. The log file is written every minute and weighs in at 2.4 GBytes composed of about 2.8 million events (we consider a single record to be an event). We extract features from the events and create a distribution to fit a model that predicts whether the network is approaching an outage, and subsequently, when the network is transitioning back to a stable state after an outage. Our predictive model needs an entire minute's worth of events to characterize their distribution and compare it against the expected distribution. This imposes a hard constraint of 60s (seconds), by which time the analytics platform must ingest the data, perform ETL, fit it to a model and extract the results.

In this paper, we describe our analytics platform at a systems level that addresses our contributions; other details like the evaluation of our predictive models, while important, are only briefly mentioned where appropriate.

4. ARCHITECTURAL APPROACHES

Our analytics spipeline is constructed from the widely used Apache open-source ecosystem; we used the following individual components in our pipeline:

- Apache Hadoop [3], version 2.7.0 with YARN [36] as a resource manager;
- Apache Spark [39], version 1.6.0. Spark was chosen over Apache Storm [35] for three reasons: (1) Spark Streaming outperformed Storm on the word count benchmark [40]; (2) default Spark libraries have native YARN support; (3) with proper code re-factoring, parts of the Spark code can be re-utilized for an eventual batch processing if the need arises;
- Apache Kafka [23], version 0.10.0.1 as the message bus, including the recently released Kafka Stream API [20];
- ElasticSearch [13] as a rich visualization engine to display results of prediction.

We experiment with the analytics pipeline on five distinct architectures (described below), each of which is implemented on a local cluster and on an Amazon Web Services (AWS) cloud to harness more resources than those available on the local cluster. The configuration of the local cluster was as follows: eight identical Mac Mini computers were used as compute blades, each with a 3.1 GHz dual-core (virtual quad-core) Intel i5 processor and 16 GB of RAM. All the hosts were rack mounted and directly connected by a commodity network switch. Each host ran Ubuntu 14.04 operating system. The virtual machines on the AWS cloud were configured as follows: 6 *m4.xlarge* VM instances (4 CPUs each, with 16GB RAM/CPU) and 5 *m4.2xlarge* VM instances (8 CPUs each, with 32GB RAM/CPU). Each VM instance used Ubuntu 14.04 operating system. Because our interest is in analyzing large datasets, we ran our experiments on the local cluster over a 100Mbps network as well as 1Gbps network to characterize the effect of network latency. On the AWS cloud, we used only 1Gbps links.

Of the five distinct architectures, one (*query-able Kafka*, c.f., §4.4) is our contribution that is compared against four others. These four are grouped into two divisions: Divison I architectures consider sending the logfile as a single large message from the Kafka consumer to the Kafka brokers while Division II architectures use variations of the well-known pattern of time window events with Spark Streaming.

4.1 Division I Architectures: A_{ks} and A_{hdfs}

The first setup is a classical stream analysis pipeline consisting of Apache Kafka and Spark Streaming; we refer to this architecture in the rest of the paper as A_{ks}. Figure 1 shows the arrangement of the 8 hosts in the local cluster for classical stream analysis. In this arrangement, five nodes: $N_{[2:6]}$ serve as Hadoop master (1 Node) and slaves (4 Nodes). Nodes N_7 and N_8 host a Kafka cluster, each running two message brokers. The Kafka brokers run with 7GB of

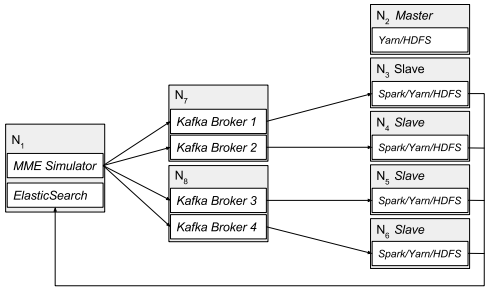


Figure 1: The A_{ks} architecture for local cluster

dedicated memory each. Each Hadoop slave node dedicates 12GB of RAM to YARN, along with 8 YARN vcores (two YARN vcores are equivalent to one physical hardware core). A Spark application receives a total of 32GB of RAM and 28 YARN vcores, and the disk swap functionality is activated. A Spark executor, the actual message consumer, gets 7GB of RAM and 6 vcores. The Spark application driver has 4GB of memory and 4 vcores, which is sufficient because the application's code does not include operations leading to the accumulation of the total message on the driver task. All operations are kept distributed on the Spark executors.

N_1 hosts an MME, which produces the messages. These messages are retrieved by a Kafka producer executing on the same node and sent out to a dedicated Kafka topic hosted on a Kafka cluster. The cluster has 4 partitions, distributed among the brokers. A partition has a replication factor of 3. The Kafka producer alternates writes on the partitions to unburden the single brokers. The Kafka broker sends the messages of interest to Kafka consumers, which execute on the slaves and are responsible for fitting the model (the analytics task) and inserting the results in ElasticSearch.

The machines in the AWS cloud are similarly arranged as in Figure 1 with the following exceptions: the Kafka cluster in the middle of the figure contains two more hosts, but we kept the number of brokers to be four as shown in the figure. In addition, there is an extra slave, for a total of 5 slaves. These extra resources, and more powerful machines on the AWS cloud allow us to push the streaming capabilities of our pipeline further than we are able to do on the local cluster.

A literature review of streaming analytics frameworks [35, 40, 9] did not consider messages comparable to the size of the messages seen in our application (2.4 GBytes). Given the size of the messages, we were interested in testing whether storing them on the HDFS distributed file system could speed up analysis compared to sending them through Kafka. To do so, we evaluate our workload on the second pipeline architecture, A_{hdfs}. The architecture for A_{hdfs} is identical to Figure 1 with the exception that the Kafka layer is completely absent, and the log messages from the MME are immediately written to HDFS upon arrival. The stored files are evaluated by a Spark task monitoring a predefined HDFS folder for new files. The hardware and software are identical to A_{ks} (Figure 1) with the only difference being that the Kafka layer (N_7 and N_8) is missing. The machines in the AWS cluster are similarly arranged with the exception of an extra slave for processing.

In both architectures, Kafka producer resides at the MME and each minute gets the 2.4 GB logfile, marshals it as a Kafka message and sends it to the Kafka broker. Kafka compresses messages sent on the network; even though the compression ratio for the logfile is 10x, nearly 205 MBytes/minute of data is sent over the network from each Kafka producer. Each minute is transmitted as a single compressed message in the Kafka message set. Normally, Kafka batches small messages and transmits them in single message set; we modify this behaviour as described in §4.4 since our message size is already large.

4.2 Division II Architectures: A_{ev} and A_{kev}

We now turn our attention to the canonical pattern in which Spark and Kafka frameworks are deployed: windowing small messages at high arrival frequencies within a specific time period [35, 40, 9]. We study two architectures that are representative of this pattern. As in Division I architectures, nodes N_7 and N_8 host a Kafka cluster, each running two message brokers; node N_1 hosts the MME and ElasticSearch. Nodes $N_{[2:6]}$ in Division II architectures, however, run two Spark master applications. In the first architecture, A_{ev}, one application is for the ETL workload and the second one for model fitting. In the second architecture, A_{kev}, the Spark ETL is replaced by an equivalent application built on the newly released Kafka KStreams API [20]. Figure 2 shows the workflow of the two architectures.

The Spark ETL application for A_{ev} (alternatively, the KStreams ETL application for A_{kev}) consumes the messages arriving from MME (panels 1, 2, 3). Recall that our target application requires a large number of messages to study their distribution for model fitting (c.f., §3). Unlike the architectures of Division I, A_{ev} and A_{kev} receive a message-at-a-time from the MME Kafka producer. As these messages arrive to the Spark (or KStream) application (panel 3), they are curated and presented back to the Kafka cluster under a different topic (panel 4). Spark buffers the messages under a windowing regiment and reassembles the data from micro-batches until an entire minute's results have been gathered. At that time, the gathered data is presented to the Spark model fitting master application (panel 5). The model results are inserted in ElasticSearch (panel 6).

We parameterize the number of messages needed to constitute enough messages corresponding to a minute; this becomes a parameter to the Spark model evaluation task to decide whether it should start computations or wait for an additional batch of data. All parameters that were required — Kafka message batch size, Spark batch size, Spark window size and window shift size — were determined through rigorous experimentation. Kafka consumers are also configured to ensure the highest possible parallelization, as in Division I: one consumer thread (panel 5) is dedicated to a single Kafka partition.

The Spark application (alternatively, the KStream application) gets 7 GB RAM and 6 vcores, each application master receives 4 GB RAM and 4 vcores. A total of 36 GB RAM and 32 YARN vcores are allocated, 4 GB RAM and 4 more vcore compared to Division I architectures because of the additional application master. The machines in the AWS cloud for Division II architectures are configured in the same manner as Division I.

4.3 Preliminary results and discussion on Division I and II architectures

Before presenting our novel on-demand query architecture, it is instructive to look at the results obtained from running our workload across the pipelines in Division I and II architectures, for both the local cluster and in the AWS cloud. The primary task performed by the Spark executors is one of prediction. The large log files arriving into the system represent data collected on the MME in minute intervals. The code executing on the Spark executors must render a prediction decision based on analyzing the contents of the log message. Specifically, for the *ProcedureIDs* of interest, over 230 fields (features) are parsed, a few extracted and a distribution of these features is fitted to a model. Conservatively, the Spark application is dedicated to monitoring a single *ProcedureID*.

All four architectures proved unsuccessful at real-time monitoring of mobile network. The failure stemmed primarily

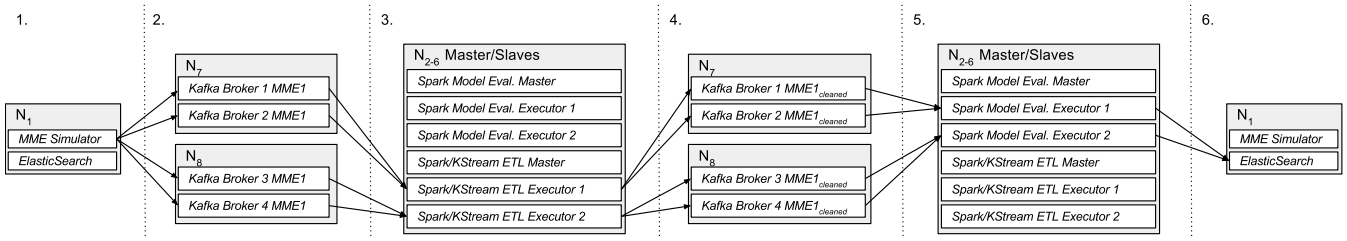


Figure 2: The A_{ev} and A_{kev} architectures on the local cluster. Panels 1,2,3: MME sends messages to the ETL tasks running on YARN (A_{ev}: Spark ETL; A_{kev}: KStreams ETL). Panels 4,5: Curated data is sent to the Spark analytics tasks via the same Kafka broker cluster under a different topic. Panel 6: The results are inserted to ElasticSearch.

from dealing with large messages and being dominated by a computational time $> 60s$ to execute the prediction model. Cumulatively, this had the effect of queuing up large messages and delaying the prediction decision. Furthermore, we observed that pushing large messages from Kafka to the rest of the pipeline components resulted in Spark dropping data partitions such that the data was unavailable for computation when needed. Because of this, we were forced to scale down the size of the messages simply to get a stable system. We found out empirically that a message size of 50 MBytes (uncompressed) was the common denominator between A_{ks} and A_{hdfs}, while a message size of 200 MBytes (uncompressed) was common to A_{ev} and A_{kev}. (For the remainder of the paper, when we mention file sizes we imply *uncompressed* files. While Kafka compresses these messages while transporting them, Spark executors perform computations on uncompressed files.)

We stress that network bandwidth is not the issue here; we ran all architectures locally on a 100 Mbps network and again on a 1 Gbps network but did not see any measurable difference in results. Computations related to our pipeline are not I/O bound, rather they are is CPU-bound. We will revisit this in greater detail in §5 where we compare results from the four architectures against our novel on-demand query mechanism. With this background, we present our novel on-demand query mechanism.

4.4 Our contribution: Novel, on-demand query-able Kafka for continuous queries

One of the strengths of query compilers in modern relational databases is their awareness of the data model and the costs for the multiple ways to access the information. Relational database managers take into account a variety of metrics to decide whether satisfying the query through indexing will be faster, or whether the result set can be gathered optimally through a linear scan [16]. Such complex decisions are absent in streaming pipelines; the no-SQL nature of such pipelines implies that the data is not even indexed. In fact, in our application the discrepancy is obvious: applying data filtering on a distributed in-memory framework implies a shuffle operation, data partitioning, network transfers, task scheduling, all as a prelude to subsequent work on fitting the data against a model. However, it would be an advantage if we could perform some filtering given that the data we are dealing with is columnar to begin with. The question is where should this filtering be done? Performing it in the Spark executors (the consumers) does not solve the problem because the executors would need access to the entire data in order to do the filtering. Performing this filtering on the MME is one option, however, this is not an attractive solution for the following reasons. One, network operators typically are shy on running extraneous processes on production-grade MMEs; the CPU and memory consumption of the MMEs is budgeted precisely for its normal workload. Thus executing extraneous processes that may cause transient spikes in CPU and memory at the MMEs is not acceptable. Second, the data produced by the MME is valuable for purposes beyond network stability. It can be used to perform other functions like endpoint identification (an Apple device versus a Samsung), monitoring subscriber quality of experience, etc. Thus it seems reasonable to move this data, as large as it is, from the MME to a data center where it could be used for various purposes. Given this discussion, the Kafka message broker provides the best location to allow filtering of the kind we envision.

Architecturally, the query-able Kafka architecture, A_{qk}, is a replica of A_{ks} (Figure 1), with the Kafka nodes $N_{\{7,8\}}$ running instances modified as described below for transport and query efficiency. Source code for modifications is available at (https://github.com/Esquive/queryable-kafka.git).

Modifications for Kafka transport efficiency: As with the A_{ks} architecture, one Spark application is dedicated to the monitoring of a *ProcedureID*. The Kafka producers marshals the entire 2.4 GByte logfile as a single message and transmit it (compressed) to the Kafka broker efficiently. To transport the messages to Kafka in an efficient way, the message format had to undergo minor modifications in regards to compression. In Kafka, messages consists of a fixed size header followed by a variable length byte array representing the key and a variable length byte array representing the value (the first 4-bytes of the key and value array contain their lengths, respectively). Figure 3-A shows a Kafka message. Messages are further aggregated into a structure called a message set. Figure 3-B shows individual messages, $M_1, M_2, ...M_n$, concatenated in a message set, each message M_i consisting of the message layout shown in Figure 3-A. The concatenated messages are compressed using a codec, which is specified in the attribute element of Figure 3-B. The resulting compressed blob becomes the "outer

Table 2: File sizes and number records/file

	Raw file size	Gzip size	No. records
$Log\ file$	2.4 GBytes	214 MBytes	2,834,242
Pid_1	448 KBytes	105 KBytes	58,133
Pid_2	150 KBytes	6 KBytes	29,729
Pid_3	9 MBytes	1 MBytes	1,078,977
Pid_4	7 MBytes	1 MBytes	861,080

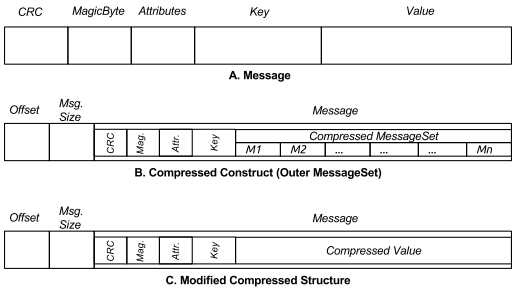


Figure 3: Kafka message format

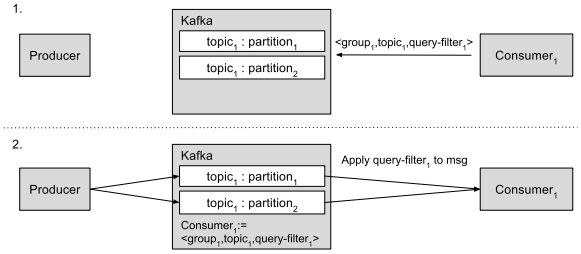


Figure 4: Query-able Kafka procedure: 1. A consumer registers with the Kafka services. 2. The consumer gets the queried topic's messages.

message" in Figure 3-B. The receiving system extracts the "outer message" and using the codec specified in the attribute, decompresses the "outer message" to retrieve each individual message M_i. Kafka bundles individual messages into a concatenated message set because individual (small) messages may not have sufficient redundancy to yield good compression ratios. Clearly, this is not true of our target application. Our messages are large, and furthermore, they compress well as shown in Table 2. Thus there appears to be relatively little value in aggregating multiple messages for our application since compressed, a 1-minute file takes about 205 MBytes. Furthermore, even if we were to aggregate multiple messages, not only would that add additional delay of waiting extra minutes for the log files to be produced, it would also lead to more complexity in the querying phase as blocks are scanned in an optimized manner to preserve message boundaries. Thus, for our query-able Kafka solution, the message format was altered, as shown in Figure 3-C, by only compressing the message value. We added a new compression indicator to the attribute element of the Kafka message, which if present, triggered our code.

In order to apply queries at the Kafka broker, a message must be decompressed first, but we wish to avoid keeping such large decompressed messages in memory for querying purposes. With our new format shown in Figure 3-C, we easily know the beginning address of the compressed message and are able to decode the message on the fly using Java IO streams to read a block, decompress it and bring only the decompressed block into memory. As each block is read in, the query is applied to the block and the result set updated. At the end of the file, the result set is transmitted to the consumer.

Modifications for Kafka query efficiency: In Kafka, consumers register with the service by enumerating a tuple consisting of <*groupdId, topic*>. For our query-able Kafka approach, the group/topic relation is extended by an additional attribute in the tuple, a *query filter*: <*groupId, topic, query-filter*>. The consumer registers with Kafka and specifies the tuple <*groupId, topic, query-filter*>, which gets saved along with the other per-consumer relevant state at the Kafka broker. Each broker in the cluster augments its native publish-subscribe capability with a limited subset of relational algebra operations, namely the *select* operation (σ) and the *project* operation (π). These operations are transmitted as parameters to the Kafka cluster when a Kafka consumer indicates interest in a specific topic. When a message corresponding to that topic arrives, the query-able Kafka cluster performs these required operations and sends the result to the consumer. In this manner, when a large 2.4 GB message needs to be sent to one or more consumers, instead of transmitting the entire message, the Kafka cluster performs the relational operations and sends the result set, which will be much smaller than the entire message.

Queries executed on an ingested message at the Kafka cluster to subset the data for a consumer are given by:

$$\pi_{field_1, field_2, ..., field_n}(\sigma_{ProcID=Pid_n})$$

The effect is that a message consumer is able to specify a rather abbreviated SQL-like query:

$$SELECT\ field_1, field_2, ..., field_n$$
$$WHERE\ ProcedureID = Pid_n$$

The $FROM$ clause is omitted since it is inherently inferred by the <*groupId, topic, query-filter*> relation. This mechanism is depicted in Figure 4.

In our application the topic is set to a unique string that identifies messages arriving from a particular MME; this topic allows multiple consumer groups to access the message on arrival. As an example, assume two *query-filters*: *query-filter*$_1$ is defined as $\pi_{field1}(\sigma_{ProcedureID} = 1)$ and *query-filter*$_2$ is defined as $\pi_{field2}(\sigma_{ProcedureID} = 2)$. Further, assume a consumer, C_1 subscribes to the Kafka cluster using the tuple <$group_1$,mme_1,*query-filter*$_1$>, that is C_1 in $group_1$ is interested in receiving notifications matching *query-filter*$_1$ from topic mme_1. Similarly, consumer C_2 in a different group from C_1 subscribes for the same topic but a different *query-filter* using the tuple <$group_2$,mme_1,*query-filter*$_2$>. When a message arrives to the Kafka cluster matching the topic mme_1, C_1 gets the column $field_1$ for all rows where $ProcedureID = 1$, while C_2 receives column $field_2$ for all rows where $ProcedureID = 2$. Because the size of this result set is much smaller than the size of the entire message, the Spark executors are able to keep up.

5. EVALUATION AND DISCUSSION

For reasons presented in §4.3, the architectures of A_{ks}, A_{hdfs}, A_{kev} and A_{kev} proved unsuccessful in their attempt at real-time monitoring of the mobile network. For each architecture the computational time to run a prediction model took much longer than a minute, the threshold after which another batch of logs would arrive. Cumulatively, this had the effect of queuing up these large messages and delaying the prediction decision.

Our evaluation proceeds in three phases. First, we conduct the experiments on a local cluster, which will serve as an empirical baseline for log files generated by one MME

and executing prediction models for one *ProcedureID*. This phase reveals the inadequacy of the first two architectures, A_{ks} and A_{hdfs}, to process files larger than 100 MBytes, and files larger than 200 MBytes for A_{ev} and A_{kev}. In the second phase, we increase the size of the MME log files on the local cluster to demonstrate the scalability of our A_{qk} solution. Finally, using the AWS cloud to provide more compute resources, we study the scalability of our system as we scale both the number of MMEs (more logfiles) and *ProcedureIDs* (more computation).

We use the following metrics for evaluation:

- Time To Completion (TTC): The difference of time (in seconds) between the message emitted by the virtual MME, and the insertion of the results into elasticSearch. TTC includes latency and measures the time needed for end-to-end handling of a logfile.
- Time from Producer to Kafka cluster (TPK): For architectures using Kafka, the time required (in seconds) to send a logfile from the MME and ingest it into Kafka. TPK is an integral part of TTC but shown separately to better estimate computational time of the Spark application.
- Main Memory Consumption (MEM): The memory allocated (in MBytes) by a process at a given time T. Throughout the evaluation, a cumulative MEM is used, which corresponds to MEM consumption on all concerned nodes that execute any Spark- or Kafka-related processes.
- CPU usage (CPU): Cumulative percentage of the CPU used by a process at a time T.

A custom Java Management Extension (JMX) [18] was developed for system resource monitoring, which writes MEM and CPU allocated to each JVM process to a logfile every second. To avoid impacting performance with excessive disk I/O, all machines on the local cluster are equipped with flash drives. On AWS a unique data store is used for logging. All Kafka and Spark related processes are monitored on every node in the cluster.

5.1 Phase I: Comparing Architectures

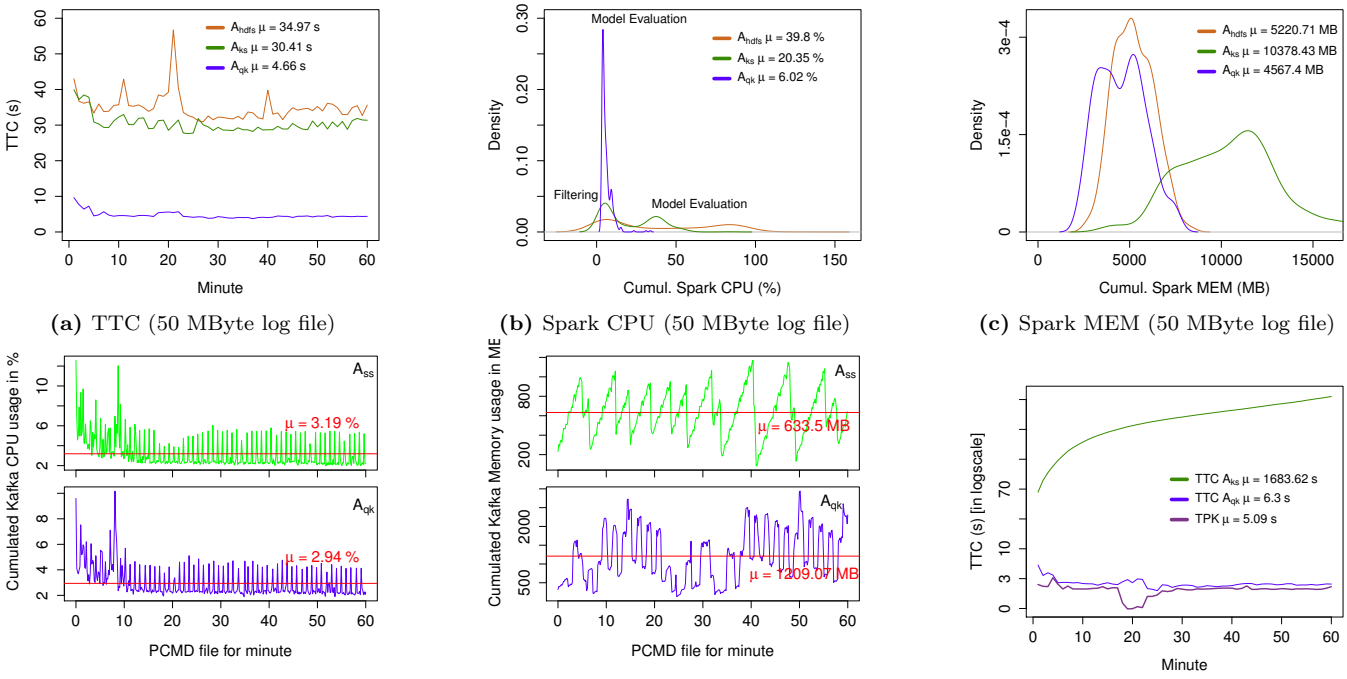
Comparing A_{ks}, A_{hdfs} and A_{qk} on the local cluster: The A_{ks} and A_{hdfs} architectures were unable to complete computations within the time constraints for our target application; A_{ks} could process logfiles that were < 100 MBytes in a timely manner, while A_{hdfs} saw a bottleneck for files > 150 MBytes. Our query-able Kafka solution (A_{kq}) can process files as large as 2.4 GBytes, but to uniformly study the behaviour of the three architectures we were forced to use the lowest common denominator size of 50 MBytes. The local cluster has sufficient resources to run a single MME as a producer and a single Spark application as a consumer performing computations for monitoring one *ProcedureID*. We chose *ProcedureID 3* since it had the most observations (c.f. Table 2).

The observed results, discussed below, are similar on the local cluster as well as on the AWS cloud, despite the increased resources AWS provides to the Spark application. Network bandwidth does not appear to play any role: outcome of the experiments is identical on a 100 Mbps and 1 Gbps networks with a savings of 2-3s on the faster network, a difference not enough to tip the balance in favour of the faster network. This strongly implies that computations are not network I/O bound but CPU bound, which is in line with the findings in Shi et al. [34].

Figure 5-A compares TTC for the three architectures processing 50 MByte files. A_{ks} and A_{hdfs} performances are almost equal but are by far outperformed by our solution, A_{qk}: A_{hdfs} has a mean TTC of 34.97s, A_{ks} 30.41s, and A_{qk} 4.66s. Our solution is about 7x (7 times) faster. Figure 5-B demonstrates the PDF for the CPU metric. Our solution, A_{qk}, is uni-modal with density concentrated around 0–10% of CPU usage, with an average usage of 6.02%. The bimodality of A_{ks} and A_{hdfs}, also observed by others [34], reflects the processing in the Spark application: one mode for filtering and the other for model evaluation. In our solution, filtering is performed by the Kafka cluster, thus the Spark application only incurs model evaluation leading to the lone peak for the A_{qk} curve. The A_{hdfs} architecture is the most demanding; CPU mean usage is 39.8%. Its curve is flatter compared to the other two because it is dominated by disk I/O, which interrupts the CPU. The cumulative CPU usage is greater than 100% because multiple cores were involved in HDFS servicing. The A_{ks} architecture is also bi-modal with a mean CPU usage of 20.35%. Our solution outperforms the other two between 3.5x to 6.5x. Figure 5-C shows the PDF for the MEM metric. A_{ks} is the most resource hungry, with usage ranging from 5 GBytes – 15 GBytes with an average of 10.3 GBytes. The range of memory required by the A_{hdfs} architecture and our solution, A_{qk}, is almost similar — 2.5 GBytes to 9 GBytes — although the peak of A_{hdfs} is higher. Our solution consumes less RAM (4.5 GBytes) than A_{hdfs} (5.2 GBytes).

The impact of our solution on system resources is in Figures 5-D and 5-E. (We do not present results for the A_{hdfs} since it does not use Kafka.) Importantly, Figure 5-D shows that adding our query-able Kafka extension has negligible impact on CPU consumption when compared to A_{ks}. The X-axis measures time, with a log file of 50 MBytes arriving every minute, for the duration of the hour that we ran the test. Our query-able Kafka solution does, however, impact memory. Figure 5-E shows that our solution consumes 3.5 GBytes of RAM versus 669 MBytes of RAM used in A_{ks}. Memory usage for A_{ks} is cyclic, constantly raising up to 1 GByte before garbage collection is triggered. By contrast, in our solution memory slowly increases up to 8G B before it starts to decrease; small fluctuations are frequent and accentuated, indicating a more recurrent garbage collection. Overall, the gradual variation of memory allocation of queryable Kafka is suggesting the system is not overloaded.

Figure 5-F shows the outcome for the Spark application on the A_{ks} architecture as we push it beyond its capability to deal with log files of 100 MBytes. For A_{ks}, it is apparent that the TTC grows linearly over the hour that we ran the system. The mean time to process a 100 MByte file in A_{ks} is 1,683.62s (28 minutes). Clearly, this does not suffice for our application! Interestingly, the TPK stays constant, which strongly suggests that the time to send a log file from the MME to Kafka is not adding to the processing delay, rather, latency is being added by the Spark application as it deals with the large log files. The curve for query-able Kafka, A_{qk}, tells a different story altogether. The mean time it takes A_{qk} to process the 100 MByte file is very low — 6.30s, suggesting that performing the filtering upstream allows the Spark application to concentrate on computation across a much smaller result set. There isn't any curve for



(a) TTC (50 MByte log file) **(b)** Spark CPU (50 MByte log file) **(c)** Spark MEM (50 MByte log file)

(d) Kafka CPU (50 MByte log file) **(e)** Kafka MEM (50 MByte log file) **(f)** TTC A_{ks} (100 MByte log file)

Figure 5: Comparing Division I architectures during Phase I

A_{hdfs} because there is no easy way to determine in HDFS when a file has been completely written to the distributed store.

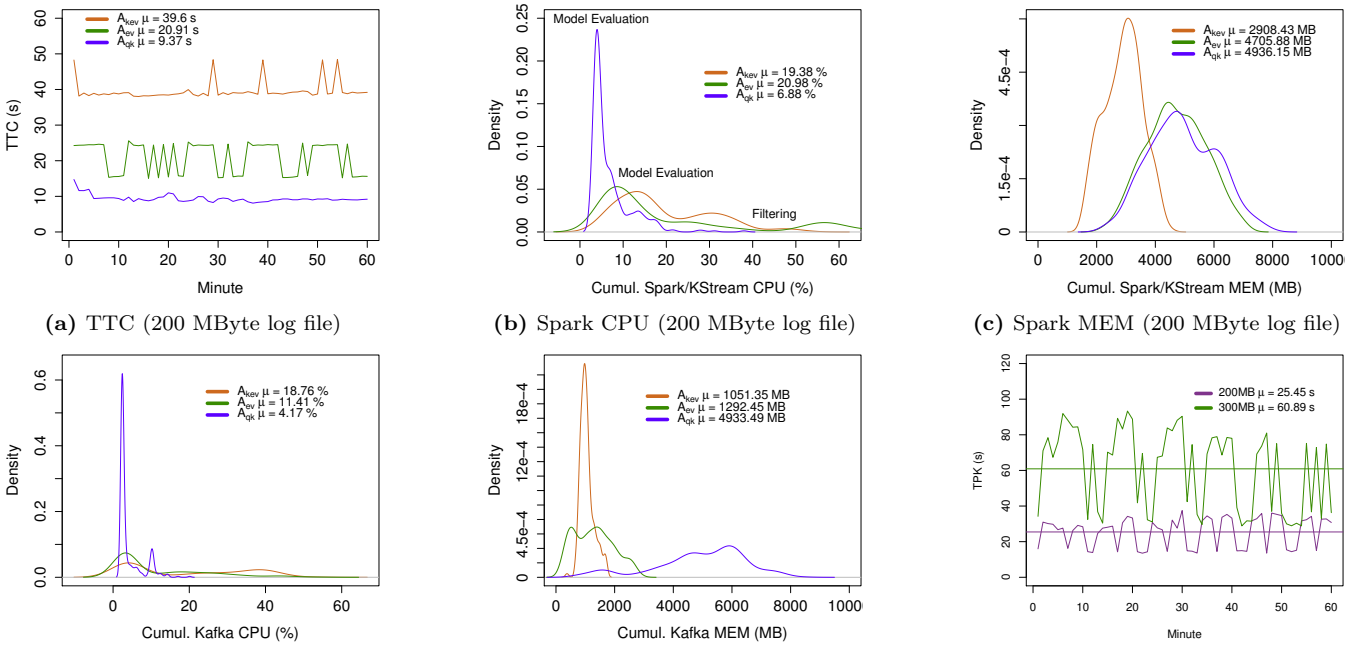
Comparing A_{ev}, A_{kev} and A_{qk} on the local cluster: The architectures in Division II proved to be more stable and better behaved than their Division I counterparts; they are able handle file sizes > 50 MBytes. Figure 6-F demonstrates that with 200 MByte files, the TPK for A_{kev} is below 60s but becomes larger when file size increases. Pushing files > 200 MBytes when using the Kafka KStreams ETL application in A_{kev} leads to unstable behaviour. The unstable behaviour is characterized by a larger standard deviation; with an unpredictable TPK, the sliding window is unable to gather enough records to constitute a stable distribution that would allow us to start the predictive task. We, therefore, use a file size of 200 MBytes for Division II architectures when comparing them against our query-able Kafka solution.

Figure 6-A shows our solution with a 2x and almost 4x advantage over A_{ev} and A_{kev}, respectively. On AWS, TTC saw minimal improvement, although this was expected on a cluster hosted on more powerful machines where each each Kafka broker had its own hard drive. We recognize, however, that allocation of hardware resources on AWS cannot be guaranteed. Figure 6-B shows the same uni-modal behaviour of the CPU metric for our solution when compared to the bi-modal behaviour of other two (for the same reasons). A_{qk}, our solution, uses the least amount of CPU. The difference in Spark MEM consumption (Figure 6-C) between Division I and II is more stark. While in Division I, our solution proved better, here A_{kev} is better (2.9 GByte). The reason is that KStreams, used by A_{kev}, contributes to low memory consumption. Regarding Kafka's resource consumption,our solution, A_{qk}, outperforms A_{ev} and A_{kev} by about 3x and 4x, respectively (Figure 6-D). This contrasts well with Division I architecture (Figure 5-D) where Kafka CPU metrics were about the same. In the Kafka MEM metric, A_{kev} performs better due to KStreams (Figure 6-E).

5.2 Phase II: A_{qk} — Scaling file size on local cluster

Among all architectures, our query-able Kafka solution is the only one that can scale to log file size of 2.4 GBytes. Therefore, we focus on its behaviour as we increase the file size with one MME and one Spark application monitoring Pid_3.

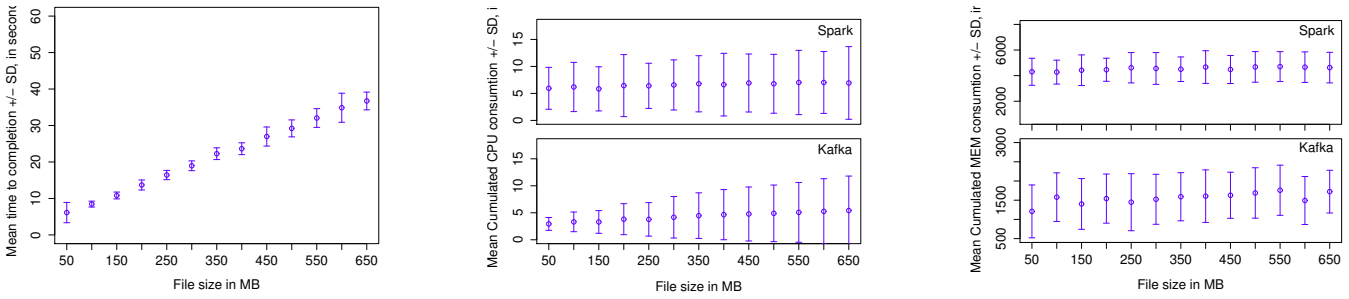
Figure 7-A shows that when the ETL filtering occurrs upstream, as in our query-able Kafka solution, Spark can handle computations fast enough to guarantee a stable system. File size is increased in 100 MByte increments starting from 100 MBytes and going to 2.4 GBytes. The time it takes for Spark computations is the difference in time between the TTC and the TPK. For a 2.4 GByte file, TTC is 77.84s and TPK is 44.33s, thus the time it takes for Spark to run the model is 33.51s, which is well within the 60s threshold our target application needs. Figure 7-B plots the CPU usage of the Kafka cluster and Spark application. For 2.4 GByte log files, Kafka shows an average CPU consumption of 17% (σ = 17%) and Spark utilizes 15% (σ = 12.7%). Figure 7-C shows the MEM metric. Spark memory remains constant at an average of 5.2 GBytes (σ = 1.1 GByte), even as the log file size increases to 2.4 GBytes. Kafka memory shows some variations. For a 2.4 GByte log file, Kafka uses, on the average, 7.3 GBytes (σ = 1.5 GBytes) memory. As the file sizes increase, Kafka memory converges to about 9 GBytes distributed across 4 brokers, or 2.3 GByte RAM per broker. This is a reasonable outcome, especially considering that the TPK is well bounded below 60s. In summary, our



(a) TTC (200 MByte log file) **(b)** Spark CPU (200 MByte log file) **(c)** Spark MEM (200 MByte log file)

(d) Kafka CPU (200 MByte log file) **(e)** Kafka MEM (200 MByte log file) **(f)** TPK A_{kev} (200 MByte log file)

Figure 6: Comparing Division II architectures during Phase I



(a) TTC, 100 to 2400MB files **(b)** CPU, 100 to 2400MB files **(c)** MEM, 100 to 2400MB files

Figure 7: Scaling logfile size during Phase II (monitoring one *ProcedureID*, Pid_3)

query-able Kafka solution is robust as the log file sizes increase, the system is stable and can keep up with rendering prediction decisions in a timely manner.

5.3 Phase III: A_{qk} — Scaling *ProcedureIDs* and MMEs on AWS

So far, all experiments involved one MME and one *ProcedureID*; we now turn our attention to how our query-able Kafka solution handles multiple *ProcedureIDs* and more than one MME sending 2.4 GByte files per minute. This experiment was conducted on the AWS cloud, where we had more resources (c.f. §4.1). In AWS, the YARN/Spark slaves are located on the first five *m4.2xlarge* instances. Since one Spark application handles one *ProcedureID*, to fit 4 Spark applications on AWS for handling four *ProcedureIDs*, each Spark executor had one less YARN vcore at their disposal.

Figure 8-A shows TTC per *ProcedureID* for four *ProcedureIDs* (c.f. Table 2) and one MME. The horizontal line at the bottom is the TPK. The first observation is that the TTC and TPK on AWS are generally higher than what was observed on the local cluster (see Figure 7-A). This is due to a number of reasons outside our control. First, unlike our local cluster, we cannot influence the placement of VMs on AWS, thus we are cannot guarantee strict bounds on the latency between the VMs hosting the MME producer, the Kafka cluster and the Spark application. Second, our local cluster always logged to a fast flash drive, but on AWS we are forced to log to a local data store, which may not be as fast. And finally, while we always ran our experiments at the same time (late night with respect to the AWS data center location), we cannot influence other activity occurring on the physical host on which executes our AWS VMs.

That said, the system on AWS is stable; to see why, we compare the time taken by the Spark application to execute the model for Pid_3 in the local cluster (§5.2, Figure 7-A) with Figure 8-A. Recall that the time it takes for the Spark computations for a model is the difference between the TTC and TPK; the mean TTC time on AWS for Pid_3 is 111.62s and the mean TPK is 75.59s giving a difference of 36.03s, very close to the value of 33.51s for the local cluster and definitely within the 60s threshold our target application requires.

The second observation is that Spark models for Pid_3, Pid_4 take a longer time than Pid_1, Pid_2; this is simply

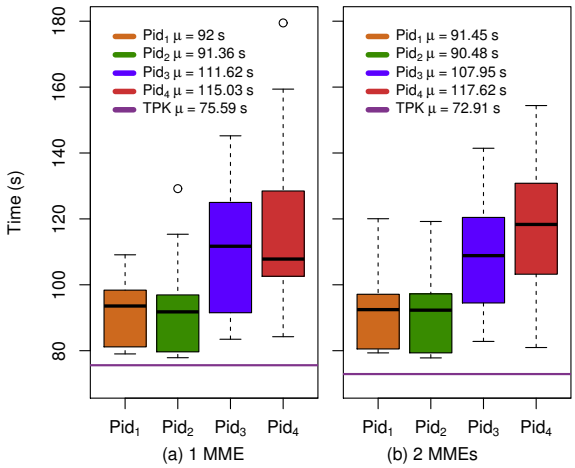


Figure 8: Scaling *ProcedureIDs* and MMEs on AWS during Phase III on 2.4 GByte logfiles

due to the fact that the former pair of *ProcedureIDs* have more observations compared to the latter pair (c.f. Table 2), consequently it takes longer to run the models. However, all models complete within the 60s time required by our target application. Finally, note that AWS offers us enough resources that we are running four Spark applications, one for each *ProcedureID* and are able to handle the load for 1 MME without any problems. We could not do this on our local cluster due to resource limitations.

Figure 8-B shows that even as we add another MME that generates an additional 2.4 GBytes log files per minute, the system is able to sustain the load in stable state. With two MMEs, the Kafka brokers are serving four consumer groups now subscribing to two topics, one for each MME. Adding a second MME has virtually no effect on the TTCs and the TPK even though twice as many log files are being processed. The boxplots of the *ProcedureIDs* between Figures 8-A and -B are similar with minor variations in the median values of Pid_4. Bolstering our argument that adding a second MME does not impact processing is Table 3; for CPU and MEM metrics on Spark and Kafka, there is very little appreciable difference between the system processing load from one MME versus two MMEs. The impact on CPU and MEM is not the number of MMEs, but rather the number of *ProcedureIDs* monitored as the analysis next shows.

Table 4 summarizes the results of Spark and Kafka analyzing 2.4 GByte files for 1 and 4 *ProcedureIDs*. For Spark MEM, there is a linear relation for moving from one *ProcedureID* to four; for n *ProcedureIDs*, Spark will use $n * 5.2$ GBytes RAM. Spark CPU, on the other hand, remains approximately constant as number of *ProcedureIDs* increases to four. However, we expect it to increase if tens of *ProcedureIDs* are monitored. Looking at Kafka resources, Kafka MEM exhibits a linear relation as well, although by a smaller multiplier ($0.5n$*9 GBytes RAM for n *ProcedureIDs*). Kafka CPU uses $n * 17\%$ for monitoring n *ProcedureIDs*. Clearly, the number of *ProcedureIDs* dictates resources usage; it is our expectation that while there are over 70 *ProcedureIDs*, practical and domain considerations limit analysis to a few.

Table 3: MEM and CPU with 4 monitored *ProcedureIDs* and two MMEs

			1 MME	2 MME
CPU (%)	Spark	mean	13.68	15.11
		σ	9.77	9.27
	Kafka	mean	69.53	67.35
		σ	69.60[1]	67.88[1]
MEM (GB)	Spark	mean	20	19.7
		σ	2.9	2.9
	Kafka	mean	20	20
		σ	7.6	7.7

[1] σ is larger than mean because the data is dispersed.

Table 4: Scalability for *ProcedureIDs*

		1 ProcedureID	4 ProcedureIDs
Spark	MEM	5.2 GBytes	20 GBytes
	CPU	15%	13.68%
Kafka	MEM	9 GBytes	20 GBytes
	CPU	17%	69.53%

Our investigations reveal that two MMEs is the limit that our cluster on AWS can handle. This limitation is imposed by Spark, not by our query-able Kafka solution, which is able to scale to more MMEs under the assumption that a handful of *ProcedureIDs* are monitored. As per the discussion on Figure 8-A, the Spark application renders a prediction decision in 36.03s. With two MMEs producing log files, it will take the Spark application about a minute to execute the models. Anything beyond two MMEs will lead to queued messages, making real-time prediction impossible. To scale out to > 2 MMEs, the Spark application can be replicated on a cluster subscribing to the Kafka topics reserved for the new MMEs. We have shown that assuming a reasonable number of monitored *ProcedureIDs*, our system scales out by allocating additional YARN/Spark clusters. This is not an unrealistic assumption; service providers are more interested in scaling the number of MMEs to cover large geographic areas than they are in monitoring more *ProcedureIDs*/MME.

6. CONCLUSION AND FUTURE WORK

To address the real-time nature of the prediction decisions required while handling GByte sized messages, we propose moving ETL upstream, specifically, in an analytics pipeline, to move the ETL to messaging layer (Kafka) thus allowing multiple speed layers (Spark application) to perform pure computational tasks. We have further demonstrated the viability of this approach through our query-able Kafka solution, which scales with the workload.

In future work, we will examine the impact on the Kafka with evolving query changes, with special emphasis on result set size and the nature of query operations vs. predicate conditions. Current supported operations are projection, selection and simple comparison predicates; we will explore the possibility of expanding the operations and predicates with the understanding that we do not want to turn Kafka into a full SQL engine, but provide it enough relational power while keeping it agile. Our query-able Kafka solution integrates querying in the internal load balancing scheme in a rudimentary fashion. In future work we will take in account optimal querying for partition placement and partition I/O.

The code for query-able Kafka modifications described in §4.4 is available in a GIT repository at https://github.com/Esquive/queryable-kafka.git.

References

[1] *50 Amazing WhatsApp Statistics*. http://goo.gl/pbqL9Y.

[2] *Apache Flink*. https://flink.apache.org.

[3] *Apache Hadoop*. http://hadoop.apache.org/.

[4] *Apache Hadoop*. https://nifi.apache.org/.

[5] S. Babu and J. Widom. "Continuous Queries over Data Streams". In: *SIGMOD Rec.* 30.3 (Sept. 2001).

[6] B. Bangerter, S. Talwar, et al. "Networks and devices for the 5G era". In: *IEEE Communications Magazine* 52.2 (2014).

[7] J. Baulier, P. Bohannon, et al. "DataBlitz: A High Performance Main-Memory Storage Manager". In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[8] E. Begoli and J. Horey. "Design Principles for Effective Knowledge Discovery from Big Data". In: *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*. 2012.

[9] *Benchmarking Apache Kafka*. https://goo.gl/3WtyXl.

[10] D. Borthakur, J. Gray, et al. "Apache Hadoop Goes Realtime at Facebook". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. 2011.

[11] C. Cranor, T. Johnson, et al. "Gigascope: A Stream Database for Network Applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003.

[12] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008).

[13] *Elastic*. https://www.elastic.co/.

[14] E. Falk, R. Camino, et al. "On Non-parametric Models for Detecting Outages in the Mobile Network". In: *IFIP/IEEE 2nd International Workshop on Management of 5G networks (5GMAN)*. To appear, May 2017.

[15] W. Fan and A. Bifet. "Mining Big Data: Current Status, and Forecast to the Future". In: *SIGKDD Explor. Newsl.* 14.2 (Apr. 2013).

[16] H. Garcia-Molina, J. D. Ullman, et al. *Database Systems: The Complete Book*. 2nd ed. Prentice Hall, 2008.

[17] V. K. Gurbani, D. Kushnir, et al. "Detecting and predicting outages in mobile networks with log data". In: *IEEE International Conference on Communications, (ICC)*. To appear, May 2017.

[18] *Java Management Extensions*. http://goo.gl/q2cNc1.

[19] W. Jonker, ed. *Databases in telecommunications: international workshop co-located with VLDB-99, Sep. 6th, 1999*. LNCS 1819. Springer, 2000.

[20] *Kafka Streams: KStream API*. http://docs.confluent.io/3.0.0/streams/.

[21] A. Karakasidis, P. Vassiliadis, et al. "ETL Queues for Active Data Warehousing". In: *Proceedings of the 2Nd International Workshop on Information Quality in Information Systems*. IQIS '05. 2005.

[22] J. Kreps. *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media, 2014.

[23] J. Kreps, N. Narkhede, et al. "Kafka: A distributed messaging system for log processing". In: NetDB, 2011.

[24] W. Lam, L. Liu, et al. "Muppet: MapReduce-style Processing of Fast Data". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012).

[25] S. Marchal, X. Jiang, et al. "A Big Data Architecture for Large Scale Security Monitoring". In: *2014 IEEE International Congress on Big Data*. 2014.

[26] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st. Greenwich, CT, USA: Manning Publications Co., 2015.

[27] G. Mishne, J. Dalton, et al. "Fast Data in the Era of Big Data: Twitter's Real-time Related Query Suggestion Architecture". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. 2013.

[28] *New Tweets per Second Record*. https://goo.gl/NCylRN.

[29] *NiFi feature request: Efficient line by line CSV Processor*. https://goo.gl/v1yYVJ.

[30] *NiFi feature request: proper CSV support*. https://goo.gl/hyV5Kg.

[31] L. Qiao, Y. Li, et al. "Gobblin: Unifying Data Ingestion for Hadoop". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015).

[32] A. Rabkin, M. Arye, et al. "Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014.

[33] N. Seyvet and I. M. Viela. *Applying the Kappa architecture in the telco industry*. https://goo.gl/0OgYTO.

[34] J. Shi, Y. Qiu, et al. "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015).

[35] A. Toshniwal, S. Taneja, et al. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014.

[36] V. K. Vavilapalli, A. C. Murthy, et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. 2013.

[37] E. J. Vergara, S. Andersson, et al. "When Mice Consume Like Elephants: Instant Messaging Applications". In: *Proceedings of the 5th International Conference on Future Energy Systems*. e-Energy 2014.

[38] F. Yang, E. Tschetter, et al. "Druid: A Real-time Analytical Data Store". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. 2014.

[39] M. Zaharia, M. Chowdhury, et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. 2010.

[40] M. Zaharia, T. Das, et al. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. SOSP 2013.

[41] L. Zhao, S. Sakr, et al. *Cloud Data Management*. Springer Publishing Company, Incorporated, 2014.