Ubiq: A Scalable and Fault-tolerant Log Processing Infrastructure

Venkatesh Basker, Manish Bhatia, Vinny Ganeshan, Ashish Gupta, Shan He, Scott Holzer, Haifeng Jiang, Monica Chawathe Lenart, Navin Melville, Tianhao Qiu, Namit Sikka, Manpreet Singh, Alexander Smolyanov, Yuri Vasilevski, Shivakumar Venkataraman, and Divyakant Agrawal

Google Inc.

Abstract. Most of today's Internet applications generate vast amounts of data (typically, in the form of event logs) that needs to be processed and analyzed for detailed reporting, enhancing user experience and increasing monetization. In this paper, we describe the architecture of Ubiq, a geographically distributed framework for processing continuously growing log files in real time with high scalability, high availability and low latency. The Ubiq framework fully tolerates infrastructure degradation and data center-level outages without any manual intervention. It also guarantees exactly-once semantics for application pipelines to process logs as a collection of multiple events. Ubiq has been in production for Google's advertising system for many years and has served as a critical log processing framework for several dozen pipelines. Our production deployment demonstrates linear scalability with machine resources, extremely high availability even with underlying infrastructure failures, and an end-to-end latency of under a minute.

Key words: Stream processing, Continuous streams, Log processing, Distributed systems, Multi-homing, Fault tolerance, Distributed Consensus Protocol, Geo-replication

1 Introduction

Most of today's Internet applications are data-centric: they are driven by backend database infrastructure to deliver the product to their users. At the same time, users interacting with these applications generate vast amounts of data that need to be processed and analyzed for detailed reporting, enhancing the user experience and increasing monetization. In addition, most of these applications are network-enabled, accessed by users anywhere in the world at any time. The consequence of this ubiquity of access is that user-generated data flows continuously, referred to as a *data stream*. In the context of an application, the data stream is a sequence of events that effectively represents the history of users' interactions with the application. The data is stored as a large number of files, collectively referred to as an input log (or multiple input logs if the application demands it, e.g., separate query and click logs for a search application). The log

captures a wealth of information that can be subsequently analyzed for obtaining higher-level metrics as well as deep insights into the operational characteristics of the application. In general, this analysis typically relies on complex application logic that necessitates joining [3], aggregation and summarization of fine-grained information. Most contemporary Internet-based applications must have backend infrastructure to deal with a constant ingestion of new data that is added to the input logs. Furthermore, this processing should be scalable, resilient to failures, and should provide well-defined consistency semantics.

The goal of Ubiq is to provide application developers a log processing framework that can be easily integrated in the context of their application without worrying about infrastructure issues related to scalability, fault tolerance, latency and consistency guarantees. Ubiq expects that the input log is made available redundantly at multiple data centers distributed globally across multiple regions. The availability of identical and immutable input logs enables the system to withstand complete data center outages, planned or unplanned. Ubiq processes the input log at multiple data centers, and is thus *multi-homed* [13]: processing pipelines are run in multiple data centers in parallel, to produce a globally synchronous output stream with multiple replicas.

Although it is often argued that data center failures are rare and dealing with them at the architectural level is overkill, at the scale at which Google operates such failures do occur. We experience data center disruptions for two reasons: (i) partial or full outages due to external factors such as power failures and fiber cuts; and (ii) shutdowns for planned maintenance. It can be argued that planned outages can be managed by migrating operational systems from one data center to another on the fly. In practice, however, we have found that such a migration is extremely difficult, primarily due to the large footprint of such operational systems; precisely checkpointing the state of such systems and restoring it without user downtime is a significant undertaking. During the past decade, we have explored numerous approaches to the operational challenge of recovering or migrating processing pipelines from an unhealthy data center to another data center. Our current conclusion is that the best recourse is to ensure that such systems are multi-homed [13].

Over the last decade, many stream processing systems have been built [1, 4, 6, 9, 10, 14, 17]. We are unaware of any published system other than Google's Photon [3] that uses geo-replication and multi-homing to provide high availability and full consistency even in the presence of data center failures. Photon is designed for applications that need state to be tracked at the *event* level, such as joining different log sources. However, this is a very resource-intensive solution for other data transformation applications such as aggregation and format conversion, where it is sufficient to track state at the granularity of *event bundles*, that is, multiple events as a single work unit. Event bundling demands far fewer machine resources, and entails different design/performance considerations and failure semantics from those of Photon. Ubiq uses different mechanisms for backup workers, work allocation, and has different latency and resource uti-

lization characteristics. See section 7 for a detailed comparison of the differences between Photon and Ubiq.

1.1 System Challenges

We next describe the challenges that must be overcome to make the Ubiq architecture generic enough to be deployed in a variety of application contexts.

- **Consistency semantics:** Log processing systems consume a continuous stream of data events from an input log and produce output results, in an incremental fashion. A critical design challenge is to specify and implement the consistency semantics of incremental processing of input events. Given the mission-critical nature of the applications that Ubiq supports, such as billing, it needs to be able to assure exactly-once semantics.
- **Scalability:** The next challenge is scalability. Ubiq needs to support applications with varying amounts of traffic on its input log. Furthermore, Ubiq needs to be dynamically scalable to deal with varying traffic conditions for a single application. Finally, Ubiq must be able to handle ever-increasing amounts of traffic. Currently, it processes millions of events per second; this is bound to increase in the future.
- **Reliability:** As mentioned earlier, Ubiq needs to automatically handle not only component failures within a data center but also planned and unplanned outages of an entire data center.
- **Latency:** The output of Ubiq is used for several business-critical applications such as analyzing advertising performance and increasing monetization. Keeping the infrastructure latency overhead to under a minute assures the effectiveness of these business processes.
- **Extensibility:** To support multiple use cases and deployments, Ubiq needs to be generic enough to be used by different applications, and to be easily integrated in a variety of application contexts.

1.2 Key Technical Insights

Here are some of the main design ideas that help Ubiq address the system challenges mentioned above:

- All framework components are stateless, except for a *small amount of globally replicated state*, which is implemented using Paxos [15]. In order to amortize the synchronization overhead of updating the global state, Ubiq batches multiple updates as a single transaction. For scalability, the global state is partitioned across different machines.
- From an application developer's perspective, Ubiq simplifies the problem of continuous distributed data processing by transforming it into processing *discrete* chunks of log records *locally*.
- Ubiq detects data center failures by introducing the notion of *ETAs*, which capture the expected response time of work units, so appropriate avoidance measures can be taken in the presence of failures.

The paper is organized as follows. In section 2, we start by presenting the overall architecture of Ubiq followed by some of the implementation details of its key components. In section 3, we describe the key features of Ubiq's design that deliver exactly-once processing, fault tolerance and scalability, in both single and multiple data centers. In section 4, we demonstrate how Ubiq can be deployed in the context of a data transformation and aggregation application. Section 5 summarizes the production metrics and performance data for a log processing pipeline based on Ubiq. In section 6, we report our experiences and lessons learned in using Ubiq for several dozens of production deployments. Section 7 presents related work and section 8 concludes the paper.

2 The Ubiq Architecture

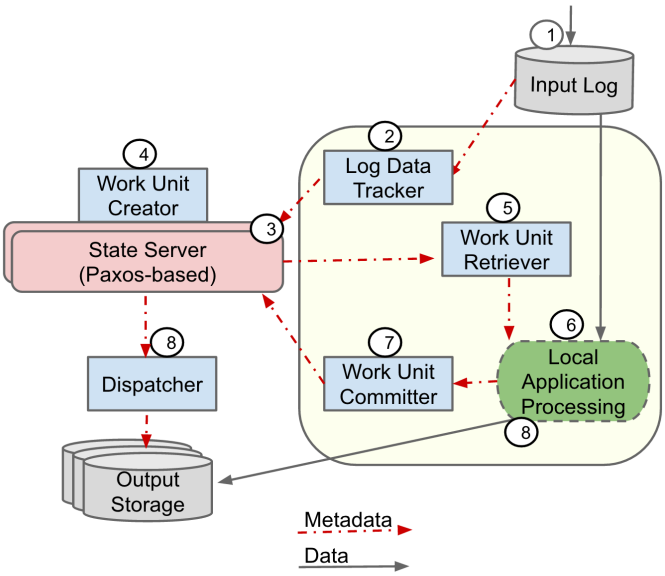

Fig. 1. Ubiq architecture in a single data center

2.1 Overview

Figure 1 illustrates the overall architecture of Ubiq in a single data center. The numbers in the figure capture the workflow in the Ubiq system, which is as follows:

1. *Input log creation:* New log events are written to the input log, which is physically manifested as a collection of files. This step is outside the scope of Ubiq, though we enumerate it here to show the end-to-end workflow.

2. *Tailing of input logs:* The first responsibility of Ubiq is to continuously monitor the files and directories associated with the input log. This functionality is performed by the *Log Data Tracker* component.
3. *Storage of metadata:* Once newly arrived data is discovered, its metadata (that is, file name and current offset) is delivered to a metadata repository, which stores the state about what has been processed and what has not (to ensure exactly-once semantics). This metadata is stored inside the *State Server*, which replicates it globally.
4. *Work unit creation:* The continuously growing log is converted into *discrete work units*, or *event bundles*, by the *Work Unit Creator*.
5. *Work unit distribution:* After the work units are created, they need to be delivered to the application for executing application logic. The *Local Application Processing* component *pulls* work units from the State Server via the *Work Unit Retriever* component.
6. *Application processing:* The *Local Application Processing* component locally applies application-specific logic such as data transformation and other business logic. See section 4 for a sample application.
7. *Work unit commitment:* Once the Local Application Processing component completes processing the work unit, it invokes the *Work Unit Committer*, which coordinates the commit of the work unit by updating the metadata at the State Server. The Work Unit Retriever and Work Unit Committer together decouple the local application processing completely from the rest of the Ubiq framework.
8. *Dispatch of results:* If the results of the local application processing are fully deterministic and the output storage system expects at-least-once semantics, the Local Application Processing component may dispatch the results to output storage directly. Otherwise, the results need to be delivered to output storage after they are committed to the State Server. This is accomplished by the *Dispatcher* component, using a two-phase commit with the output storage.

As described above, the Ubiq framework is relatively simple and straightforward. The challenge, as described in Section 1.1, is to make this system strongly consistent, scalable, reliable and efficient.

2.2 Ubiq Architecture in a Single Data Center

Expectations for Input Log Data: Ubiq expects input files in multiple data centers to reach eventual consistency byte by byte. For example, new files may get added or existing files may keep growing. When created redundantly, the corresponding files in different regions may have different sizes at any time, but should become identical at some point in the future. If a file has size S_1 in one data center and size S_2 in another and $S_1 < S_2$, the first S_1 bytes in the two files must be identical.

State Server: The State Server is the globally replicated source of truth about log processing status, and the center of communication among all other

Ubiq components. It is implemented using a synchronous database service called PaxosDB [8] that performs consistent replication of data in multiple data centers using Paxos [15], a distributed consensus protocol. It stores the metadata about what has been processed and what has not. For each input log file and offset, it maintains three possible states:

- Not yet part of a work unit
- Already part of a work unit in progress
- Committed to output storage

It maintains this information efficiently by merging contiguous byte offsets in the same state; that is, maintaining state at the granularity of <filename, begin_offset, end_offset>.

All other framework components interact with the State Server. The State Server receives information about newly arrived data from the Log Data Tracker, uses this meta-information to create work units that will be delivered via the Work Unit Retriever to the Local Application Processing component, and commits the work units that have been completed. The metadata information stored at the State Server is critical to ensure the exactly-once semantics of Ubiq. The State Server suppresses any duplicate information received from the Log Data Tracker. All metadata operations, such as work unit creation, work retrieval by the Work Unit Retrievers, and work commitment by the Work Unit Committers, are executed as distributed transactions using atomic read-modify-write on the underlying storage at the State Server.

Log Data Tracker: The primary task of the Log Data Tracker is to discover growth of data in the input logs, which occurs in two ways: new input log files, and increases in the size of existing files. The Log Data Tracker continuously scans input directories and registers new log filenames in the State Server with their current sizes. It also monitors the sizes of existing files and informs the State Server when new bytes are discovered.

The Tracker runs independently in or near each input logs data center and only notifies the State Server of updates in its local logs data center. Since the State Server de-duplicates the information received from the Tracker, the design for the Tracker is simplified, to provide at-least-once semantics. Every file is tracked by at least one Tracker worker. Every update is retried until successfully acknowledged by the State Server.

Work Unit Creator: The Work Unit Creator runs as a background thread inside the State Server. Its goal is to convert the continuously growing log files into *discrete work units*, or *event bundles*. The Work Unit Creator maintains the maximum offset up to which the file has grown at each of the input logs data centers. It also stores the offset up to which work units have been created in the past for this file. As it creates new work units, it atomically updates the offset to ensure that each input byte is part of exactly one work unit. In order to prevent starvation, the Work Unit Creator prioritizes bytes from the oldest file while creating work units. The Work Unit Creator also tries to ensure that a work unit has chunks from several different files, as these could be read in parallel by the application.

Work Unit Retriever and Work Unit Committer: The goal of these two framework components together is to decouple the local application processing completely from the rest of the Ubiq framework. The Work Unit Retriever is responsible for finding uncommitted work units in the State Server. It delivers these work units to the Local Application Processing component (whenever the latter *pulls* new work units) and tracks this delivery through the global system state. Once the Local Application Processing component completes processing a work unit, it requests a commit by invoking the Work Unit Committer. This initiates an atomic commit, and if successful, the global system state is updated to ensure that the data events in the completed work unit will not be processed again. On the other hand, if the commit fails, the work unit will be retried again to ensure exactly-once semantics.

Dispatcher: If the results of an application are deterministic and the output storage system expects at-least-once delivery, the Local Application Processing component can directly deliver the results to output storage system. Otherwise, a dedicated framework component, the Dispatcher, delivers the results of the Local Application Processing to output storage. The Dispatcher needs to perform a two-phase commit between the State Server and the output storage system to ensure exactly-once semantics. Ubiq currently supports dispatching to Mesa [12] and Colossus (Google's distributed file system). Ubiq has a generic API that can be extended to support more output storage systems in future.

Garbage Collector: Once a work unit is dispatched to the output storage, a background thread in the State Server is responsible for garbage-collecting the work unit and all the metadata associated with it. This thread also garbage-collects input filenames once these get older than a certain number of days (e.g., d days) and they are fully processed. The State Server guarantees that if it receives an input filename (from the Log Data Tracker) with a timestamp older than d, it will drop the filename. The Log Data Tracker only tracks files at most d days old.

2.3 Ubiq Architecture in Multiple Data Centers

So far we have focused on the Ubiq design in the context of a single data center. Figure 2 shows the detailed architecture of the Ubiq system deployed over two data centers.

Replication of critical state: In Ubiq, the critical component that must remain consistent across data centers is the *global system state maintained at the State Server*. In particular, the global state information must be synchronously maintained with strong consistency across multiple data centers to ensure that we do not violate the exactly-once property of the log processing framework. This is accomplished by using PaxosDB, as described in the previous section on the State Server. All metadata operations, such as work creation, work retrieval by the Work Unit Retrievers, and work commitment by the Work Unit Committers, are executed inside State Server as distributed transactions across multiple data centers globally. In order to amortize the overhead of individual transactions, we use several system-level optimizations such as batching multiple transactions.

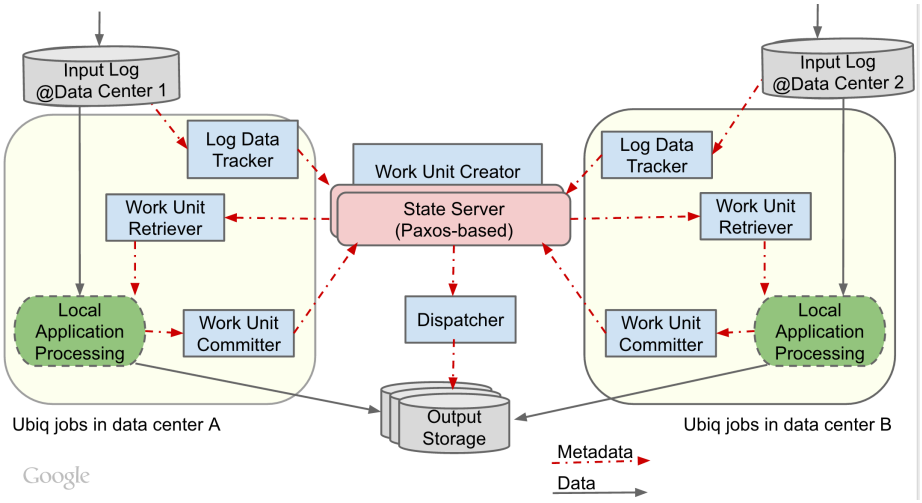

Fig. 2. Ubiq architecture in two data centers

De-duplication of input from multiple logs data centers: As mentioned earlier, Ubiq expects input bytes in multiple data centers to reach eventual consistency byte by byte. The Log Data Tracker in each data center independently tracks growth of data in the corresponding input log data center. The Work Unit Creator unifies data from multiple input log data centers to create *global* work units. It does this by maintaining a key-value data structure inside the State Server. The *key* is the basename of a file (i.e., name of the file without the path). Inside the *value*, it stores metadata about the file in all input logs data centers. If the input bytes are available in only one input data center, it will mark this in the work unit so that the work unit is preferably processed only in a nearby data center. The State Server assigns work units uniformly amongst all healthy data centers, or proportionally to the fraction of resources provisioned by the user for local application processing in each data center.

Replication of output data: It is possible that the data center containing the output of the Local Application Processing component may go down before the results are consumed by output storage. In order to handle this, Ubiq must be able to either roll back a committed work unit to regenerate the output or replicate the output of the Local Application Processing component into another data center, before committing the work unit into the State Server. If the application's business logic is non-deterministic and output storage has partially consumed the output, rollback is not an option. In order to address this, Ubiq provides a *Replicator* component as a first-class citizen. The Replicator copies a file from the local filesystem to one or more remote filesystems in other data centers.

Preventing starvation: Even though Ubiq does not provide any hard ordering guarantees, it ensures that there is no starvation. Each framework component prioritizes the oldest work unit. For example, the Work Unit Creator adds

the oldest bytes when creating work units; the Work Unit Retriever retrieves the oldest work unit, and so on.

3 Ubiq System Properties

3.1 Consistency Semantics

Depending upon the nature of the underlying applications, processing of input data events can be based on (i) at-most-once semantics; (ii) at-least-once semantics; (iii) exactly-once semantics; or (iv) in the extreme case, no consistency guarantees. Given that Ubiq has to be generic to be used in multiple application contexts, it provides *exactly-once semantics*. Supporting this consistency guarantee introduces significant synchronization overhead; however, our experience is that a large class of applications, especially those with financial implications (e.g., billing advertisers, publisher payments, etc.), warrant exactly-once processing of data events in the input log. As mentioned in section 2.3, Ubiq achieves exactly-once semantics by de-duplicating input from multiple logs data centers, executing all metadata operations on input byte offsets as distributed Paxos transactions inside the State Server, and ensuring that there is no starvation.

Note that for applications to leverage the exactly-once guarantees of Ubiq, they must write code that does not have side effects outside the Ubiq framework (e.g., updating a global counter in an external storage system). The Ubiq design does not provide any ordering guarantees; it restricts the processing logic of input events to be independent of each other. However, Ubiq ensures that input events are not starved.

3.2 Fault Tolerance in a Single Data Center

Here is how Ubiq handles machine failures in a single data center:

- *All components in Ubiq are stateless*, except the State Server, for which the state is stored persistently. Within a single data center, every Ubiq component can be executed on multiple machines without jeopardizing the correctness of the system. Hence, each component is relatively immune to machine failures within a data center.
- The State Server leverages PaxosDB [8] for fault tolerance. If an application is running entirely in a single data center, Ubiq enables running multiple PaxosDB group members as replicas in the same data center to handle machine failures.
- To handle *local application processing failures*, we use a notion called *Estimated Time of Arrival*, or *ETA*, to capture the expected amount of time processing a work unit should take. Violations of ETAs are clues that something might be wrong in the system. To handle local application processing failures within a single data center, we define a *local ETA* with each work unit. When a Work Unit Retriever obtains a work unit W from the State Server, the State Server

marks it with an ETA of t time units. When a mirrored Work Unit Retriever approaches the State Server within t time units, W is blocked from distribution. On the other hand, if the Retriever requests work after t time units have elapsed, then W becomes available for distribution. This allows backup workers in the same data center to start processing the work unit if the original worker is unable to process before the local ETA expires. Duplicate requests to commit a work unit (either because two workers were redundantly assigned the work unit or the request to commit got duplicated at the communication level due to timeout) are suppressed at the State Server since only one of them will be accepted for commit and the others will be aborted.

3.3 Fault Tolerance in Multiple Data Centers

As mentioned in [13], a data center is in *full outage* mode if it is completely unresponsive. A data center in *partial outage* mode is responsive but its performance / availability may be significantly degraded. Although both partial and full outages are handled by migrating workloads from a malfunctioning data center to a healthy one, there are some major differences in how such workload migration takes effect.

Impact of full data center outage: Google has dedicated services that continuously monitor full data center outages and notify interested systems; Ubiq learns about full data center outages proactively using these external signals. In the normal case, Ubiq assigns new work units uniformly amongst all the active data centers. During a full data center outage, Ubiq stops assigning any work unit to the unhealthy data center. Existing work units assigned to the unhealthy data center are immediately re-assigned to one of the healthy data centers. The entire workload is handled by the remaining healthy data centers as soon as the full outage occurs. Assuming that the healthy data centers are provisioned to handle the entire load, there is no impact on end-to-end latency.

Impact of partial data center outage: Unlike full data center outages, there are no direct signals or monitors to detect partial data center outages. Hence, we need to build mechanisms inside Ubiq to deal with partial outages. As mentioned in Section 3.2, the notion of local ETA allows us to have backup workers in the same data center. However, in the case of a partial data center outage, backup workers in the unhealthy data center may continue to process the same work unit, leading to starvation of work units. In order to prevent this from happening, we have another ETA inside the State Server, known as the *data center ETA*. We give a work unit to one data center by default and set the data center ETA to T minutes. If the work unit is not committed within T minutes, it is made available to another data center. This ensures that if one processing data center goes down or is unable to complete the work within the specified SLA, backup workers in the other processing data center will automatically take over the pending workload. Therefore, when a partial data center outage occurs, the workload migration does not take effect immediately, but needs to wait for the timeout of the data center ETA. The healthy data center picks up the timed-out work units from the slow data center only after the data center ETA expires. In

practice, the data center ETA is set to be an order of magnitude larger than the local ETA. This ETA timeout contributes to increased latency. If an application does not want to see these latency spikes, it can set a lower value for the data center ETA at the expense of higher resource cost.

Note that the existence of the data center ETA does not remove the need to have a local ETA. Local application processing often performs intermediate local checkpoints of partial data processing. Having a local ETA allows a backup worker in the same data center to resume processing from these checkpoints.

The consequence of this design is that the overall Ubiq architecture is resilient to both partial and full data center outages; furthermore, it can be dynamically reconfigured from N data centers to N' data centers, which makes our operational task of running log processing pipelines in a continuous manner 24×7 significantly more manageable.

3.4 Scalability

As mentioned above, all components in Ubiq, with the exception of the State Server, are stateless. This means that they can be scaled to run on multiple machines without compromising on consistency.

To ensure that the *State Server* does not suffer from scalability bottlenecks, the configuration information uses a key concept to partition the work among multiple machines: input filenames are hashed to an integer domain, which is configured in terms of a certain number of partitions; i.e., $\langle integer \rangle$ MOD $\langle number\_of\_partitions \rangle$. Each machine is responsible for a single partition.

To make the State Server design extensible, we need to allow the partitioning information to be dynamically re-configured without bringing the system down. We do this by maintaining configuration information for different time ranges. Each input filename encodes an immutable timestamp based on a global time server utility (TrueTime [11]) to ensure that the timing information is consistent across all filenames and across all regions. The configuration of the State Server has a time range associated with it. That is, it may be that from $\langle$5:00AM $Today\rangle$ to $\langle$5:00PM $Today\rangle$ the State Server has 10 partitions whereas from $\langle$5:01PM $Today\rangle$ onward it has 20 partitions. During the transition, the State Server decides which partition mechanism to use based on the encoded timestamp, until it is safe to transition to a new configuration.

3.5 Extensibility

Finally, Ubiq's design is extensible. From an application developer's perspective, Ubiq simplifies the problem of continuous distributed data processing by transforming it into processing discrete chunks of log records locally. Ubiq's API can be used by any application-specific code, and hence can be easily integrated in a variety of application contexts. The application developer only needs to provide the log processing code and some configuration information such as the input log source filename(s), the number of workload partitions, and number of data centers.

4 Data Transformation & Aggregation: An Application using Ubiq

We now describe how the Ubiq framework is used to deploy a critical application at Google. The goal of this application is to continuously transform and aggregate log events into higher-level measures and dimensions, and to materialize the results into downstream storage systems such as Mesa [12], an analytical data warehousing system that stores critical measurement data. As mentioned in section 2, Ubiq separates the processing responsibilities into (i) a common framework that focuses on incremental work management, metadata management, and work unit creation; (ii) specialized *local application processing*, which focuses on the application logic required to process a new set of input events. This application logic has the following responsibilities:

- Transforming input events: The local application processing transforms the input events based on application needs. Such transformation may involve data cleaning and standardization, splitting a single event into multiple rows destined to multiple tables in the underlying database, annotating each input event with information from databases, applying user-defined functions, executing complex business logic, etc.
- Partially aggregating input events: Although downstream systems can perform aggregation internally, given the massive size of input, it is much more resource-efficient if the input is partially aggregated. The local application processing performs a partial GROUP BY operation on each bundle of input events.
- Converting data into requisite storage format: Input data is stored in a row-oriented format that needs to be transformed into a columnar data layout.

Note that this application could be built using the Photon [3] architecture as well. However, it would be very resource-intensive to store state at the event level.

Figure 3 illustrates the above application using the Ubiq framework. The application developer is responsible only for the development of the subcomponent that encodes the application logic, the Data Transformer & Aggregator. This subcomponent relies on a well-defined API that is provided to the application developer for interfacing with the Ubiq components. This deployment uses the *Replicator* component of Ubiq since the underlying business logic is non-deterministic.

5 Production Metrics

Deployment setup: Ubiq in production is deployed in a highly decentralized manner (see Figure 4). As shown in the figure, the input logs are made available redundantly in at least two regional data centers, e.g., data centers 1 and 2. The Ubiq pipelines are active in at least three data centers, e.g., A, B, and

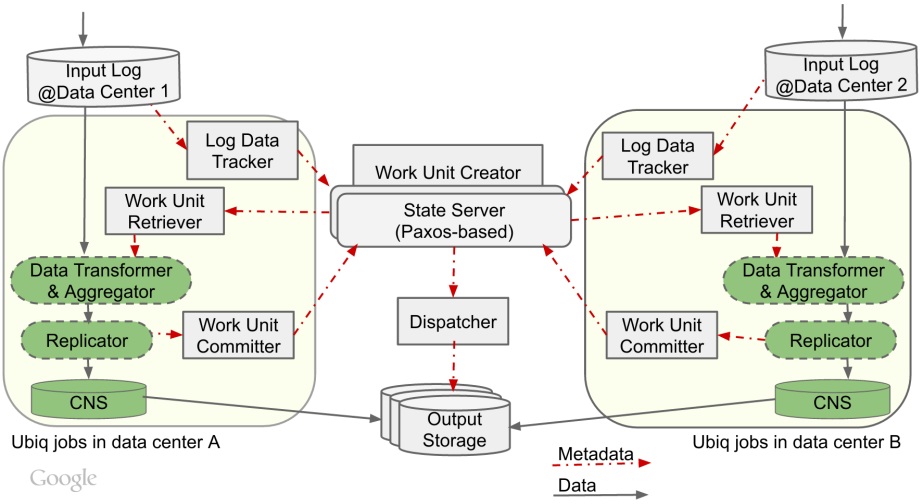

Fig. 3. Data Transformation and Aggregation: An application using Ubiq

C. To preserve data locality, data centers A and C are close to 1 while data center B is close to 2. The global system state, although shown as a centralized component, is in general actively maintained in a synchronous manner in at least 5 different data centers. If data center B, for example, experiences either a partial or complete outage, then data centers A and C will start sharing the workload without any manual intervention and without any breach of SLA. This assumes that there are enough resources available at data centers A and C for scaling Ubiq components for additional workloads.

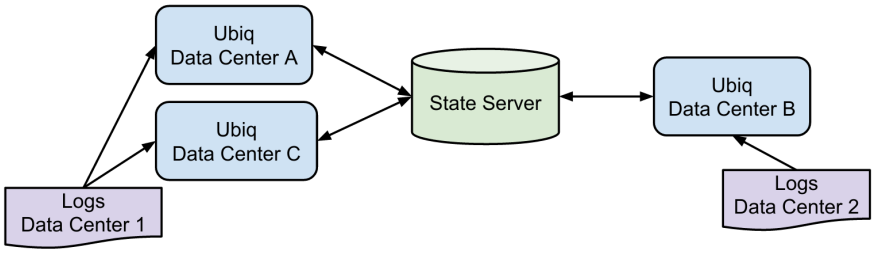

Fig. 4. A distributed deployment of Ubiq over multiple data centers

We next report some of the critical production metrics to highlight the overall performance characteristics of the Ubiq framework. At Google, Ubiq is deployed for dozens of different log types, which effectively means that we have several

dozen different pipelines with different data rates being continuously processed. The scale of a typical pipeline is on the order of a few million input events per second, producing several million output rows per second. The metrics reported in this section correspond to two such pipelines.

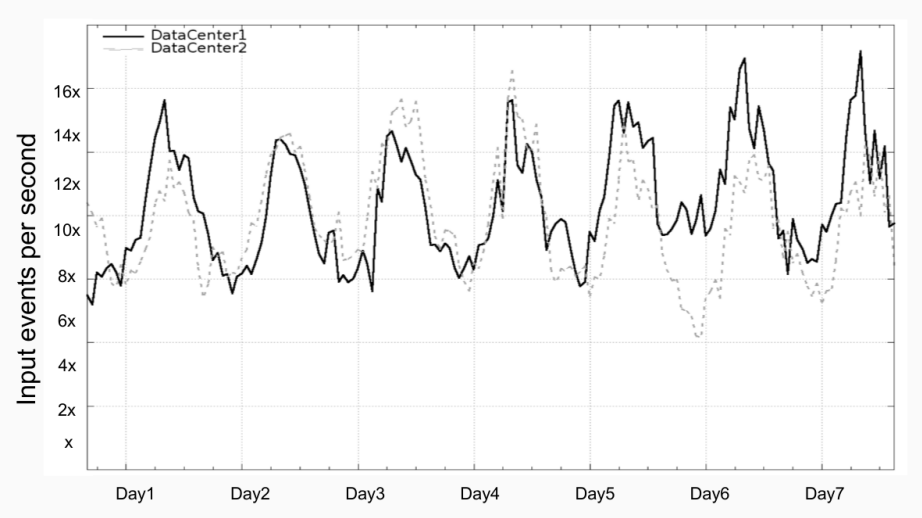

Fig. 5. Throughput during normal period

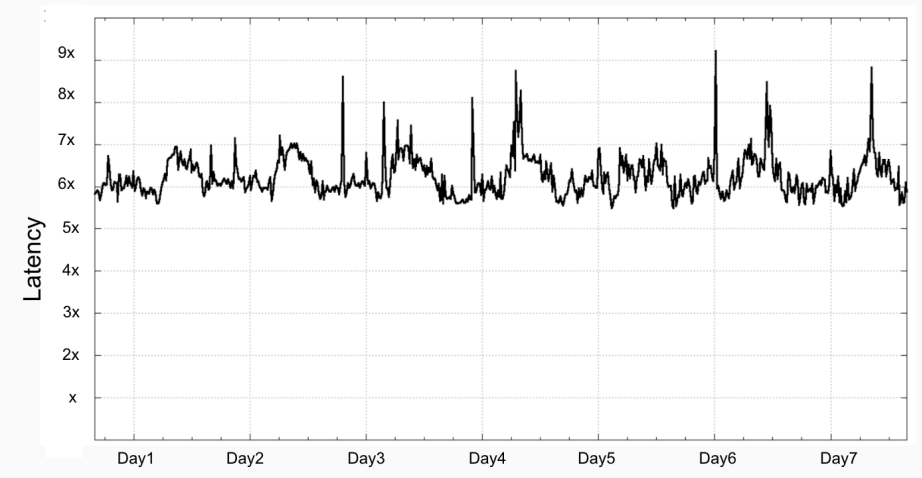

Fig. 6. Latency during normal period

Throughput and latency during normal periods: Figure 5 illustrates the throughput that is processed by Ubiq at each data center. One observation we make is that the load is evenly distributed across both data centers. Figure 6 illustrates the 90^{th} percentile latency associated with processing the input events for the same log type during the same period as Figure 5. These latency numbers correspond to the difference between the time when the bytes are first tracked by Ubiq and the time when the bytes are dispatched to output storage. Note that the latency corresponds to global results produced at both data centers. Based on our internal instrumentation, when there is no application processing in the pipeline, latency is approximately under a minute for the 90^{th} percentile. All the additional latency therefore comes from the application processing of this particular log type.

Impact of full data center outage: Figures 7 and 8 analyze the system behavior in the presence of a full data center outage. In Figure 7, we observe that one of the data centers experiences a full outage, resulting in the increased workload at the other data center. However, the 90^{th} percentile latency metrics in Figure 8 demonstrate that latency is not adversely impacted during workload migration. As explained in Section 3.3, Ubiq gets an external signal for a full data center outage, and immediately shifts the entire workload to the healthy data centers. Each data center is provisioned to handle the complete load. Note that there is a huge spike in latency for a very brief period during the full data center outage. This is because of a big increase in the number of input events (due to minor upstream disruptions).

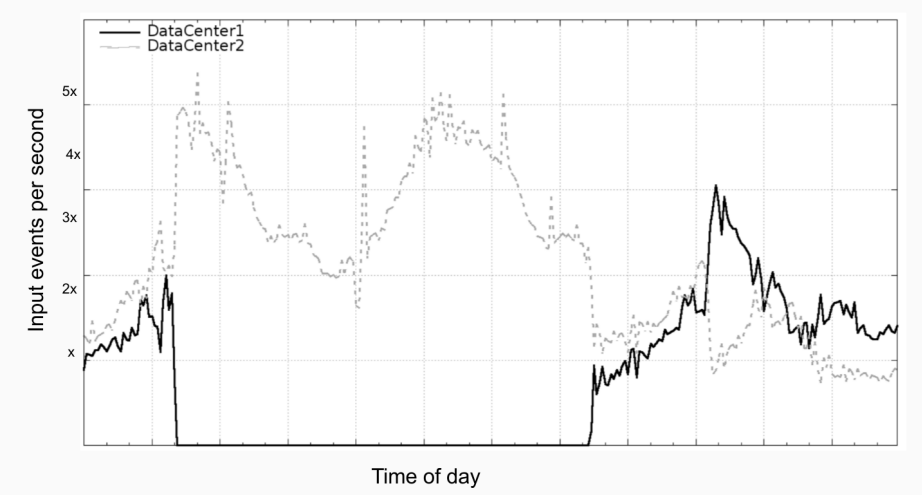

Fig. 7. Throughput during full data center outage

Impact of partial data center outage: Figures 9 and 10 depict the behavior of the system in the presence of a partial data center outage. In Figure 9,

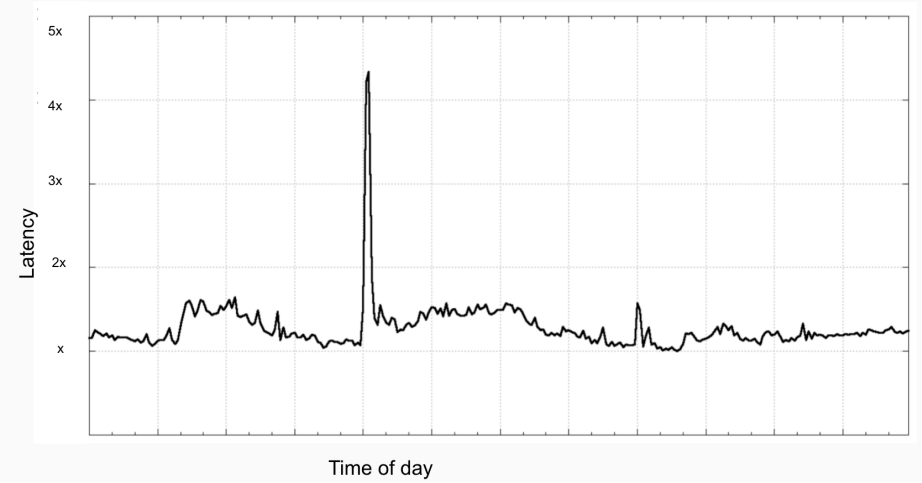

Fig. 8. Latency during full data center outage

around 5:30am, one of the data centers experiences a partial outage and as a result, its throughput declines sharply, while the other data center picks up the additional load. Figure 10 reports the latency for the 90^{th} percentile: indeed, during the partial outage, the latency in processing input log events increases considerably. As explained in section 3.3, this is because the shift of the workload to the healthy data center happens after the data center ETA expires.

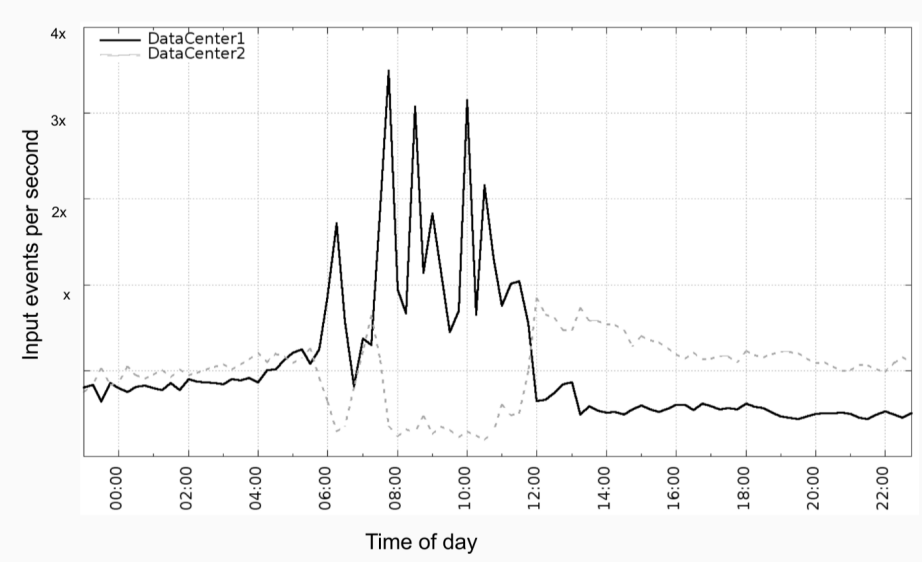

Fig. 9. Throughput during partial data center outage

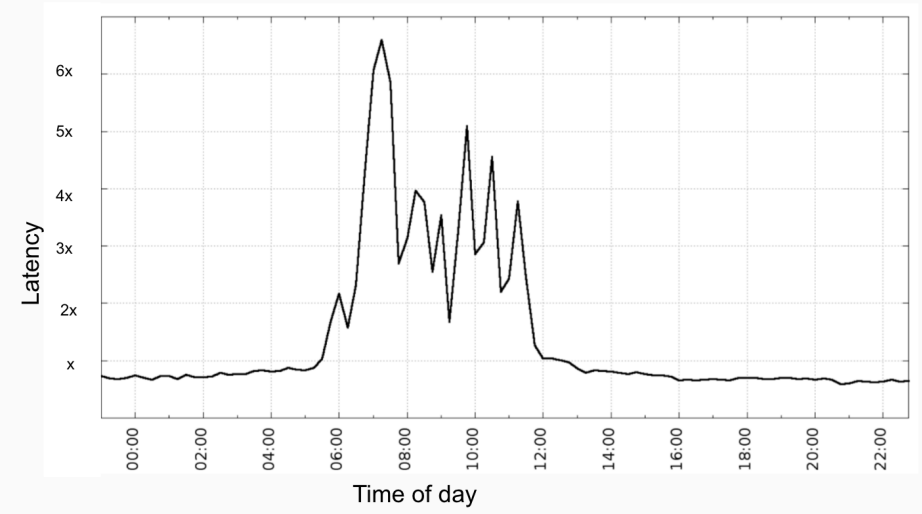

Fig. 10. Latency during partial data center outage

In summary, we note that both partial and full data center outage are handled transparently by the Ubiq framework and no manual intervention is required.

6 Experiences and Lessons Learned

In this section, we briefly highlight the main lessons we have learned from building a large-scale framework for continuous processing of data streams in a production environment. One key lesson is to prepare for the unexpected when engineering large-scale infrastructure systems since at our scale many low-probability events do occur and can lead to major disruptions.

Data corruption: As an infrastructure team, we account for software and hardware failures of individual components in the overall system design. However, a major challenge arises in accounting for data corruption that occurs because of software and hardware failures much lower in the stack. The scale at which Ubiq runs increases the chance of seeing these bugs in production. Also, there may be bugs in the business logic inside the Local Application Processing component or in an upstream system. This may cause the Local Application Processing component embedded in Ubiq to fail an entire work unit due to a handful of bad events.

We have built several solutions to address the issue of data corruption. The first approach is to provide a detailed reporting tool that allows the application developer to identify the exact byte range of the input work unit where the problem occurs. In addition, we have gone one step further, where a failed work unit with diagnostics about corrupted byte range is automatically split into multiple parts: the corrupted byte range and the uncorrupted byte ranges. A byte

range consists of <filename, begin_offset, end_offset>. The uncorrupted ranges are queued as new work units and the corrupted byte range is reported back to the Ubiq clients for further investigation. This ensures that all the uncorrupted byte ranges associated with the original work unit can be processed successfully.

Automated workload throttling: Even though Ubiq's design is highly scalable, in practice, bottlenecks arise within the system due to external factors. For example, when the system requests additional machine resources to scale the local application processing component, if there is a delay in provisioning, there will be workload buildup within Ubiq. If no measures are taken this can adversely impact the health of the overall system or may unnecessarily initiate multiple resource provisioning requests to Google's Borg system [16], resulting in under-utilization of resources later on. In order to avoid such problems, we have built monitoring and workload throttling tools at every stage of Ubiq to throttle work generation from the upstream components when Ubiq finds that downstream components are overloaded.

Recovery: Even though Ubiq itself replicates its state in multiple data centers, for critical business applications we guard against the failure of the entire Ubiq pipeline by keeping additional metadata inside the output storage system for each work unit. This metadata tracks the list of input filenames/offsets used to generate the output. In the case of Ubiq failure, we can read this metadata from both the output storage system and the input logs to bootstrap the state of Ubiq in the State Server. In theory, if the majority of the machines on which a Paxos partition is running go unhealthy, this can lead to corrupted state at the State Server. In practice, a more likely cause of Ubiq failure is an accidental bug in the code that leads to inconsistent state in the State Server.

7 Related Work

During the past decade, a vast body of research has emerged on continuous processing of data streams [1, 2, 7, 9, 10]. Most of these systems are research prototypes and the focus has been to develop declarative semantics for processing continuous queries over data streams. Over the past few years, the need for managing continuous data has become especially relevant in the context of Internet-based applications and services. Systems such as Storm [6], Samza [5], Spark Streaming [17], Apache Flink [4] and Heron [14] are available in the open-source domain for continuously transforming granular information before it is stored. However, none of these systems are *multi-homed*; they operate in a single data center and hence are vulnerable to data center outages.

The only published system that is geo-replicated and provides both multi-homing and strong consistency guarantees even in the presence of data center failures is Google's Photon [3]. The main distinction between Ubiq and Photon is that Photon is targeted for *event-level* processing whereas Ubiq supports processing *multiple events as a single work unit* (i.e., event bundles). The difference in processing granularity between Photon and Ubiq leads to the following differences in design and performance tradeoffs:

- *Principal use cases:* One canonical application for Photon is *log joining*, where each log event is joined independently with other log sources and output as a new augmented event. In contrast, Ubiq works best for applications like *partial aggregation* and *data format conversion* where multiple events are processed together to generate new output, which can then be efficiently consumed by downstream applications.
- *Resource efficiency:* In contrast with Photon, Ubiq does not need to maintain global state at the event level, and hence requires significantly fewer machine and network resources to run. Although it is feasible to trivially perform every data transformation, event by event, using Photon, it would be wasteful in machine resources for those data transformations where event-level state information is not necessary.
- *Backup workers:* The bundle processing in Ubiq allows for global, ETA-based work unit distribution (to all processing sites), which results in near-zero duplicate work. In contrast, all processing sites in a Photon system need to read all events and the de-duplication takes place later.
- *Failure semantics:* For applications where the processing of each event may fail and needs to be retried (e.g., transient lookup failures to an external backend), Ubiq would have to fail the entire work unit if the number of failed events is beyond a threshold, which renders the idea of work-unit splitting prohibitively expensive. By contrast, even in the worst-case scenario, if every alternate event fails processing and needs to be retried, Photon would process the successful events and commit them to the output since it maintains event-level state.
- *Work allocation:* As Ubiq is based on bundle-level granularity, it is much easier to allocate work to local application processing using a *pull-based* mechanism. Photon, on the other hand, leverages *push-based* work allocation.
- *Latency:* Ubiq incurs noticeably higher latency (on the order of tens of seconds) compared to Photon (on the order of seconds). Ubiq needs to wait to create event bundles while Photon does not need to incur this cost. Pull-based work allocation also contributes to higher latency in Ubiq. As a result of the different strategy for backup workers, partial data center outages impact overall latency in Ubiq while Photon handles partial data center outages seamlessly without any latency impact.

8 Concluding Remarks

In this paper, we have presented the design and implementation details of an extensible framework for continuously processing data streams in the form of event bundles. We illustrated how the Ubiq framework is used in real production applications in Google. One of the key aspects of Ubiq's design is to clearly separate the system-level components of the framework from the application processing. This extensibility allows a myriad of applications to leverage the processing framework without duplication of effort. Ubiq's extensibility has proven to be a powerful paradigm used by dozens of applications, even though it was originally envisioned to simplify the operational issues for a handful of very large

customers. Another key feature of Ubiq is that it provides exactly-once semantics. Although there are no ordering guarantees, exactly-once semantics make application logic considerably simpler: application developers do not have to complicate processing logic to handle missing or duplicate data. To deal with the high variability of input data rates, Ubiq's design is highly scalable and elastic: additional resources can be provisioned or removed dynamically without impacting the operational system. Component failures at the data center level are handled by redundantly processing the work units in a staggered manner. Finally, the multi-homed design of Ubiq makes it effective in dealing with full and partial data center outages transparently, without any manual intervention. In the future, we plan to develop a service-oriented architecture for Ubiq for more effective accounting, access control, isolation, and resource management. We are also exploring the use of machine learning models for fine-level resource management and predictive control.

References

1. D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
2. D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.
3. R. Ananthanarayanan et al. Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams. In *SIGMOD*, pages 577–588, 2013.
4. Apache Flink. `http://flink.apache.org`, 2014.
5. Apache Samza. `http://samza.apache.org`, 2014.
6. Apache Storm. `http://storm.apache.org`, 2013.
7. A. Arasu et al. STREAM: The Stanford Stream Data Manager. In *SIGMOD*, page 665, 2003.
8. T. D. Chandra et al. Paxos Made Live - An Engineering Perspective. In *PODC*, pages 398–407, 2007.
9. S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing. In *SIGMOD*, page 668, 2003.
10. J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.
11. J. C. Corbett et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.
12. A. Gupta et al. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB*, 7(12):1259–1270, 2014.
13. A. Gupta and J. Shute. High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads. In *BIRTE*, 2015.
14. S. Kulkarni et al. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, SIGMOD '15, pages 239–250, 2015.
15. L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
16. A. Verma et al. Large-scale cluster management at Google with Borg. In *EuroSys*, pages 18:1–18:17, 2015.
17. M. Zaharia et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, pages 423–438, 2013.