Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams

Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, *
Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko *,
Deomid Ryabkov, Manpreet Singh, Shivakumar Venkataraman
Google Inc.
{ananthr, vbasker, sumitdas}@google.com,
{agupta, jianghf, tianhao}@google.com, areznich@mpi-sws.org,
{rojer, manpreet, shivav}@google.com

ABSTRACT

Web-based enterprises process events generated by millions of users interacting with their websites. Rich statistical data distilled from combining such interactions in near real-time generates enormous business value. In this paper, we describe the architecture of Photon, a geographically distributed system for joining multiple continuously flowing streams of data in real-time with high scalability and low latency, where the streams may be unordered or delayed. The system fully tolerates infrastructure degradation and datacenter-level outages without any manual intervention. Photon guarantees that there will be no duplicates in the joined output (at-most-once semantics) at any point in time, that most joinable events will be present in the output in real-time (near-exact semantics), and exactly-once semantics eventually.

Photon is deployed within Google Advertising System to join data streams such as web search queries and user clicks on advertisements. It produces joined logs that are used to derive key business metrics, including billing for advertisers. Our production deployment processes millions of events per minute at peak with an average end-to-end latency of less than 10 seconds. We also present challenges and solutions in maintaining large persistent state across geographically distant locations, and highlight the design principles that emerged from our experience.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems — Distributed databases; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Design, Performance, Reliability

Keywords

Continuous Streams, Fault-tolerance, Paxos, Stream joining

*Work done during internship with Google.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Joining data streams has received considerable attention in the past two decades [24, 25] due to its importance in numerous applications (e.g. IP network management, telephone fraud detection), where information arrives in the form of very high-speed streams that need to be processed online to enable real-time response [9].

With the explosive evolution of the Internet and the World Wide Web in the last several years, the need for similar technologies has grown multifold as web-based enterprises must process events generated by millions of users interacting with their websites. Users all over the world visit Google's web pages on a daily basis, issuing web-search queries, browsing search results, and clicking on ads (advertisements) shown on result pages.

Distilling rich statistical data from web user interactions in near real-time has a huge impact on business processes. The data enables advertisers to fine-tune their bids, budgets and campaigns, and vary the parameters in real-time to suit changing user behavior. It provides immediate feedback to advertisers on the effectiveness of their changes, and also allows Google to optimize the budget spent for each advertiser on a continuous basis.

To provide real-time statistical data, we built a system, called *Photon*, that can relate a primary user event (e.g. a search query) with subsequent events (e.g. a click on an ad) within seconds of the occurrence of such events. Photon joins multiple continuous streams of events using a shared identifier to combine the related events.

Let us illustrate the operational steps in Photon with an example of joining a click and a query event:

- When a user issues a search query (using terms such as "buy flowers") at google.com, Google serves ads to the user along with search results. The web server that serves the ad also sends information about this event to multiple *logs-datacenters*, where the data is stored persistently in the Google File System [14] (GFS). The logged data includes information such as advertiser identifier, ad text, and online ad auction parameters. This data is later used to generate reports for advertisers, perform quality analysis, etc. Each query event is assigned a unique identifier known as *query_id*. Figure 1 shows the query event happening at time t_1.
- After receiving the results of the search query, the user may click on one of the ads. The ad click request is sent to a backend server while the user is being forwarded/redirected to the advertiser's website. The

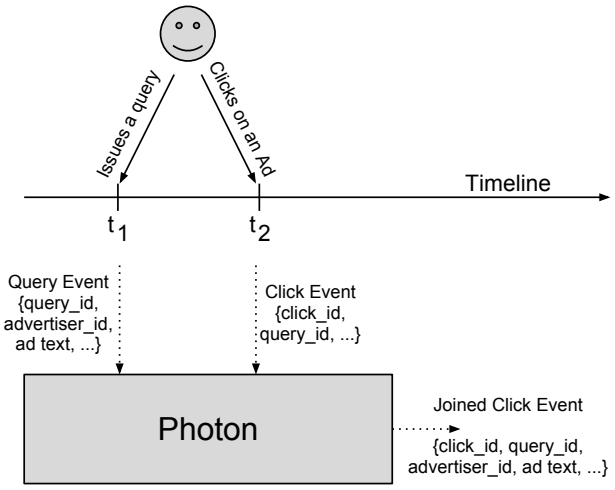

Figure 1: Joining query and click events in Photon

click event is also logged and copied to multiple logs datacenters. The logged data includes information of the ad that the user clicked, and is used to bill advertisers. The click event contains the query_id of the corresponding query. Each click event is assigned a unique identifier known as *click_id*. Figure 1 shows the click event happening at time t_2.

Continuing with the example, when a click event is shipped to logs datacenters, Photon joins the click event with its corresponding query event based on query_id. As part of the join, Photon copies all the details from the query event into a new joined click event (as shown in Figure 1). Photon reads input event streams from log files on GFS in different datacenters and writes joined events to new output log files on GFS in multiple datacenters. Joined logs are used to generate rich statistical data, combining the information from queries, ads and clicks.

Note that individual logs do not contain all the information necessary for generating rich statistical data. For example, when we compile statistics on a click event, we need access to the complete information recorded in the corresponding query event, such as other ads that were shown in the query result page, in addition to the ad recorded by this click event. It is possible to embed the query information in the Click URL sent to the user. However, sending extra data would increase latency of response and degrade user experience, as well as increase exposure to critical data. Moreover, URL length has an upper limit, thus restricting the amount of information we can propagate about a query. Hence, it is infeasible to include all the required information in the click event.

1.1 Problem Statement

Formally, given two continuously growing log streams such that each event in the *primary* log stream contains a unique identifier, and each event in the *foreign* log stream contains the identifier referring to an event in the primary log stream, we want to join each foreign log event with the corresponding primary log event and produce the joined event.

In relational database (RDBMS) terms, we can consider the two log streams as two tables with a foreign-key constraint [7]. The primary log stream is analogous to the primary table, and the foreign log stream corresponds to the foreign table. The stream joining problem is essentially an inner join between these two log streams. More specifically, the click logs mentioned above represent a foreign table that will be joined with the primary table (i.e., query logs). Hence, we refer to click events as foreign events, and query events as primary events, with the query_id as the key used in the joins. The resulting events are referred to as joined events. We use the generic terms in the rest of the paper, but we may use clicks and queries as illustrative examples.

The main goal of Photon is to perform continuous stream joining in real-time. We use this framework to join many different event streams at Google.

1.2 System Challenges

While building Photon to join continuous data streams, we face these challenges:

- ***Exactly-once semantics***: The output produced by Photon is used for billing advertisers, reporting revenue to investors in Wall Street, and so on. Photon must guarantee that a single click is never processed twice, since this leads to double-charging the advertisers. On the other hand, Google loses money for every click which is not processed by Photon. In practice, to meet the business needs, we require Photon to join 99.9999% events within a few seconds, and 100% events within a few hours. These requirements imply that Photon must provide: a) at-most-once semantics at any point of time, b) near-exact semantics in real-time, and c) exactly-once semantics eventually.

- ***Automatic datacenter-level fault-tolerance***: Datacenters regularly face different forms of outages. Some of these are planned (such as a software update or hardware replacement) and others are unplanned (like a power failure or network fiber cut). Service interruptions can last from a couple of minutes to a few days or even weeks. In this environment, a system designed to operate in a single datacenter suffers from serious drawbacks. After a failure, manual effort is required to setup the system in another datacenter, which can be tedious and error-prone. Reconstructing the state of the system can take hours, thus adversely affecting the overall availability of the system. In case of transient hiccups, it is difficult to decide whether to initiate the expensive effort of migrating the system to another datacenter, or try to ride out the outage, again impacting the system availability. Some distributed systems like GFS [14] and Bigtable [6] are excellent at handling a limited number of machine failures. But they are not designed to handle large-scale datacenter-level outages, and thus developers are responsible for designing their applications to handle outages gracefully. Given its direct impact on revenue, business needs mandate Photon to be a system with a very high degree of fault-tolerance that can automatically handle even a datacenter-level outage, with no manual operations and no impact on system availability.

- ***High scalability***: Not only does Photon need to handle millions of events per minute today, it must also be able to handle the ever-increasing number of events in the future.

- ***Low latency***: The output of Photon is used to compute statistics for advertisers and publishers on how their marketing campaigns are performing, detecting invalid clicks, optimizing budget, etc. Having Photon perform the join within a few seconds significantly improves the effectiveness of these business processes.

- ***Unordered streams***: Our primary stream (i.e. query) contains events approximately sorted by the timestamps of the events. However, the foreign stream (i.e. click) is typically not sorted by the query timestamp, since clicks can be arbitrarily delayed relative to the queries. This makes it very hard to apply window join algorithms [24, 25] proposed in literature.

- ***Delayed primary stream***: A click can only be joined if the corresponding query event is available in the logs datacenter. Logically, the query event always occurs before the corresponding click. However, the servers generating click events and query events are distributed throughout the world (to minimize end-user latency), and click and query logs are shipped independently to logs datacenters. The volume of query logs is orders of magnitude more than the volume of click logs, thus, it is not uncommon for a subset of query logs to be delayed relative to the corresponding click logs. Photon needs to be able to join whenever the query is available. This makes Photon different from standard RDBMS where a foreign key must always exist in the primary table.

1.3 Our Contributions

The key contributions from this paper are:

- To the best of our knowledge, this is the first paper to formulate and solve the problem of joining multiple streams continuously under these system constraints: exactly-once semantics, fault-tolerance at datacenter-level, high scalability, low latency, unordered streams, and delayed primary stream.

- We present challenges and solutions in maintaining persistent state across geographically distributed locations. While using commodity hardware resources, special attention is given to improving fault-tolerance and increasing throughput.

- The solutions detailed in this paper have been fully implemented and deployed in a live production environment for several months. Based on this real-world experience, we present detailed performance results, design trade-offs, and key design lessons from our deployment.

The rest of this paper is organized as follows. In section 2, we discuss how to store the critical state of our system in multiple datacenters while addressing all the systems challenges discussed above. Section 3 presents the detailed design and architecture of the Photon system. Section 4 describes our production deployment settings and measurements collected from the running system. Section 5 lists the lessons learnt from building Photon. Sections 6 and 7 describe related research, and summarize future work and conclusions.

2. PAXOS-BASED ID REGISTRY

The simplest way to achieve fault-tolerance on commodity hardware is through replication [22]. Extending this principle to a large collection of machines, we can withstand datacenter-level outage by running the same system in multiple datacenters in parallel. This approach has been applied to almost all web search servers and ad servers at Google to render the systems resilient to datacenter failures with no discontinuity in the level of service. Load balancers automatically redirect each user request to the closest running server, where it is processed without the need to consult any other server.

To provide datacenter-level fault-tolerance, Photon workers in multiple datacenters will attempt to join the same input event, but workers must coordinate their output to guarantee that each input event is joined at-most-once. The critical state shared between the workers consists of the set of event_ids (e.g. click_id) that have already been joined in the last N days. This state is stored in the *IdRegistry*. The constant N is determined by evaluating the trade-off between the cost of storage, and the cost of dropping events that are delayed by more than N days. Note that the system should always discard events delayed by N days as there is no way to detect duplicates among such events, and our goal is to avoid double-charging the advertisers.

Before writing a joined event to output, each worker verifies whether the event_id already exists in the IdRegistry. If the identifier exists, the worker skips processing the event. Otherwise, the worker attempts to write the event_id into the IdRegistry. The worker must successfully insert the event_id into the IdRegistry before it can write the joined event to output the logs. The IdRegistry must guarantee that once an event_id is written to it, subsequent requests to write the same identifier will fail.

The key requirements for the IdRegistry can be summarized as follows:

- *Fault-tolerance at datacenter-level*:
  The IdRegistry must be synchronously replicated to multiple datacenters so that its service is always available even when there is an outage in one or more datacenters. The maximum number of simultaneously tolerated datacenter outages should be configurable.

- *Read-modify-write transactions*:
  Workers must be able to perform conditional commits such as writing an event_id in the IdRegistry only if it does not exist in the IdRegistry yet.

Given the requirements of fault-tolerance and strong consistency, we implemented the IdRegistry using Paxos [18]. Paxos is a consensus algorithm executed by a set of replicas (e.g workers), to agree on a single value in the presence of failures. The algorithm guarantees synchronous replication of submitted values across the majority of replicas.

Figure 2 shows the high-level design of Photon to illustrate the use of the IdRegistry. We deploy the same Photon pipeline in multiple datacenters. Each pipeline processes all the events (e.g., clicks) present in the closest logs datacenter — at Google, logs are written to multiple geographically distributed datacenters. Each pipeline keeps retrying until the event is joined (i.e., written to the IdRegistry), which guarantees that each event is processed at least once by the system with minimal loss. To ensure that each event is processed at-most-once, the IdRegistry stores identifiers for the

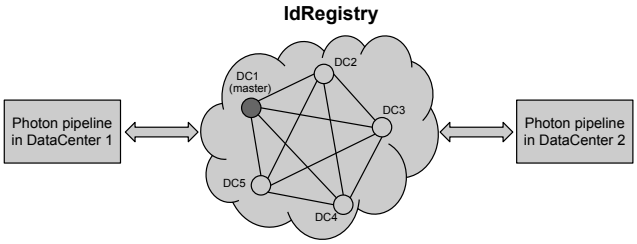

Figure 2: Photon pipeline running independently in two datacenters with the IdRegistry storing the global state synchronously in multiple datacenters.

events (e.g., click_id) that have been joined in the last N days. Since the global list of already joined events is stored in the IdRegistry, the Photon pipelines in multiple datacenters can work independently without directly communicating with each other.

2.1 IdRegistry Server Architecture

To implement an efficient IdRegistry, we use an in-memory key-value store that is consistently replicated across multiple datacenters using Paxos. By storing the id of an event as a key, the IdRegistry can quickly identify if an event is already joined or not. If the event_id already exists, it can fail the worker that tries to commit the same event_id. If the event_id does not exist, the IdRegistry does a Paxos commit to insert the new key with the condition that the key is still absent (i.e., using a read-modify-write transaction).

The IdRegistry is built on top of PaxosDB [4], a fault tolerant key-value in-memory store that uses the Paxos consensus algorithm. The PaxosDB implementation executes the consensus algorithm repeatedly for every incoming value to guarantee that each replica has access to the same sequence of submitted values. Each replica constructs its in-memory key-value store by processing the same sequence of submitted values. In other words, the in-memory key-value store at each caught-up replica is consistent. PaxosDB also guarantees that at most one of the group members will be the *master* at any given instant of time. Only the master can submit updates to Paxos. If the master dies, PaxosDB automatically elects a new master.

Figure 3 shows the architecture of a single IdRegistry server. The interaction between Photon workers and the IdRegistry follows a client-server model. Photon workers take the role of clients by sending two kinds of requests: (1) a lookup request to check if an event_id has already been committed; and (2) a conditional commit request to insert an event_id if and only if it is not present. The Remote Procedure Call (RPC) handler threads in the IdRegistry server accept incoming RPCs from clients. The handler is very lightweight in that it simply adds the input requests into an in-memory queue. A background thread dequeues requests, performs transactions on PaxosDB, and executes RPC callbacks which send responses to the client.

2.2 Scalable IdRegistry

To ensure that the outage of one relatively large geographical region does not affect the IdRegistry, we place IdRegistry replicas in different geographical regions such that replicas from more than one geographical region have to agree before a transaction can be committed into PaxosDB.

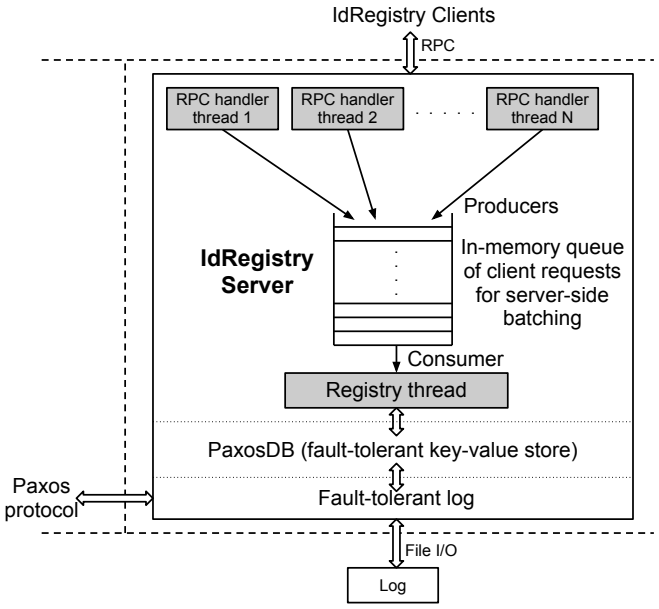

Figure 3: Architecture of a single IdRegistry server shard.

The downside of requiring such isolation zones is that the throughput of the IdRegistry will be limited by network latency. Based on typical network statistics, the round-trip-time between different geographical regions (such as east and west coasts of the United States) can be over 100 milliseconds. This would limit the throughput of Paxos to less than 10 transactions per second, which is orders of magnitude fewer than our requirements—we need to process (both read and write) tens of thousands of events (i.e., key commits) per second.

Although each client may be able to bundle multiple key commits into one RPC request, when there are a large number of clients, client-side batching is not always effective. We describe two mechanisms to achieve high scalability with the IdRegistry: server-side batching and sharding.

2.2.1 *Server-side Batching*

IdRegistry replicas are located in different geographical regions, hence a Paxos commit can take up to 100 milliseconds due to the round-trip latency. The latency mainly comes from the inherent network topology and not due to the amount of data we are sending around. In fact, each Paxos commit only needs to send a small amount of event-level data. Based on this observation, we implemented server-side batching to improve the throughput of each IdRegistry server. The key idea is to combine multiple event-level commits into one bigger commit. This is similar to the classic database technique of group commit [23].

As shown in Figure 3, the IdRegistry server has a single background thread (called registry thread) that dequeues client RPC requests, translates them to PaxosDB transactions, and sends the RPC response. The registry thread dequeues multiple requests and batches them into a single PaxosDB transaction.

Within a batch of requests, the registry thread performs application-level conflict resolution. Consider the case where multiple requests try to insert the same event_id into the Id-

Registry. If the event_id does not already exist, the registry thread will insert the event_id into PaxosDB, and respond with success to one client which marks the event as joined; other clients will receive a failure message which must drop the event. We take advantage of multi-row transactions in PaxosDB (updating multiple key/value pairs atomically with condition checks) to implement this behavior.

Assuming the size of an RPC for one event_id is less than 100 bytes, we can easily batch thousands of RPC requests into a single PaxosDB transaction. We do not enforce a minimum batch size, hence there is no tradeoff on latency.

2.2.2 Sharding

Although server-side batching improves the throughput of the IdRegistry significantly, it still fails to meet our requirements. The motivation for *sharding* comes from this observation: in the IdRegistry, events with different ids are processed independently of each other. To take advantage of the event_id independence, we partition the event_id space handled by the IdRegistry into disjoint *shards* such that event_ids from separate shards are managed by separate IdRegistry servers. The IdRegistry servers run in parallel and can accept requests for event_ids deterministically assigned to them. Figure 4 shows client requests being sent to different server shards, depending on their event_ids. The shard number is computed using a deterministic hash of the event_id, modulo the total number of shards. Note that each Photon worker can still process events from multiple shards; it can send RPCs to multiple IdRegistry server shards in parallel. This sharding approach ensures high scalability, and allows the IdRegistry to achieve the throughput we need.

Dynamically changing the number of shards

As the number of logged events increases, we need to add more shards to the IdRegistry. However, while adding more shards, simply changing the number of shards in the modulo can violate the deterministic property of the hash, and that can result in the same event_id to be mapped to a different shard number after increasing the shards. This is unacceptable because it will lead to duplicate events in the output logs. Thus, we need a deterministic mapping mechanism that allows dynamic changes of the number of shards while preserving mapping of existing event_ids across shards — the mapping mechanism must support backward compatibility.

To solve the above problem, we use a timestamp-based sharding configuration, which defines the number of shards to use at any given instance of time. We associate a timestamp with an event_id, and require that the clock skew of two timestamps is bounded by S seconds (using a global TrueTime [8] server, see section 3.1 for details). If the current time is t_1, we choose a future time $t_2 > t_1 + S$ and specify that events with timestamp less than t_2 should use hash(event_id) modulo the number of shards before increasing the shards, while the events with timestamp $>= t_2$ should use hash(event_id) modulo the number of shards after increasing the shards. This guarantees that we will deterministically compute the shard number for a given event_id.

As an example, if the IdRegistry has 100 shards, the sharding configuration will be:

$start\_time : 0, end\_time : \infty, number\_of\_shards : 100$

If we increase the number of shards from 100 to 120, current time is 5000, skew is 200, our sharding configuration will change to:

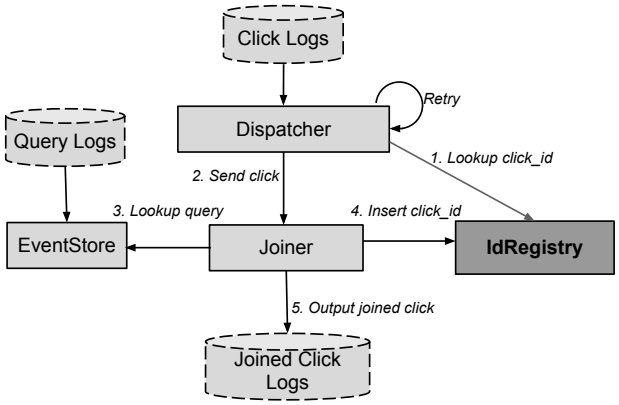

Figure 5: Components of pipeline in a single datacenter

$start\_time : 0, end\_time : 5200, number\_of\_shards : 100$
$start\_time : 5200, end\_time : \infty, number\_of\_shards : 120$

We store the sharding configuration inside PaxosDB to ensure that all the IdRegistry server shards and clients share the same configuration. Note that number of shards can be decreased in a similar way.

2.3 Deleting Old Keys

We can only keep a finite number of event_ids in the IdRegistry. Since each event_id is timestamped, the IdRegistry can delete ids older than N days. We refer to N as *garbage collection threshold*.

A background thread in each IdRegistry server periodically scans the old keys, and deletes them from the IdRegistry if the timestamp of these keys is older than the garbage collection threshold. The garbage collection thread runs only at the master IdRegistry replica. To guard against time differences between the different IdRegistry servers when the master changes, we store the garbage collection boundary timestamp in the IdRegistry servers, one timestamp for each IdRegistry server shard. This boundary timestamp is periodically updated by another thread using TrueTime [8] and we make sure it never goes back. The garbage collection thread only removes event_id entries whose timestamps are smaller than the recorded boundary timestamp.

If a client tries to insert or lookup an event older than the boundary timestamp, the IdRegistry sends failure response with a special error code, and the client skips processing the event.

3. SINGLE DATACENTER PIPELINE

As mentioned in Section 2, we deploy the same Photon pipeline in multiple datacenters. We now describe the architecture of a pipeline at a single datacenter, as shown in Figure 5. In the following examples, we use click and query logs as inputs, but the architecture applies to any similar event streams.

There are three major components in a single datacenter pipeline: *the dispatcher* to read clicks continuously and feed them to the joiner; the *EventStore* to provide efficient lookup for queries; and the *joiner* to find the corresponding query

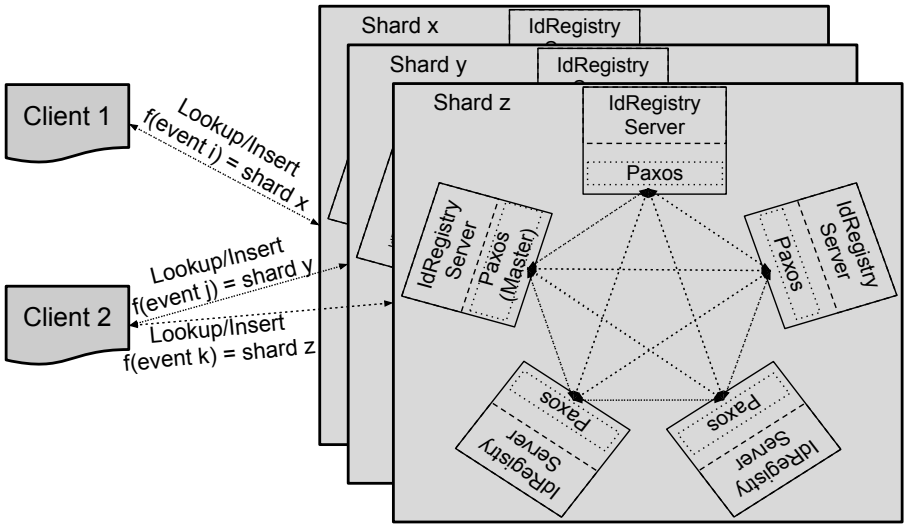

Figure 4: Sharding the IdRegistry to scale throughput

for a click using EventStore, deduplicate using the IdRegistry, and then generate joined output logs.

The sequence of steps to join a click with its corresponding query into a joined event are:

1. The dispatcher consumes the click events from the logs as they come in, and issues a lookup in the IdRegistry. If the click_id already exists in the IdRegistry, the dispatcher assumes that the click has already been joined and skips processing the click.
2. If the click_id does not exist in the IdRegistry, the dispatcher sends the click to the joiner asynchronously and waits for the response. If the joiner fails to join the click (say, due to a network problem, or because of a missing query event), the dispatcher will keep retrying by sending the click to another joiner instance after some backoff period. This guarantees at-least-once semantics with minimum losses.
3. The joiner extracts query_id from the click and does a lookup in the EventStore to find the corresponding query.
4. If the query is not found, the joiner sends a failure response to the dispatcher so that it can retry. If the query is found, the joiner tries to register the click_id into the IdRegistry.
5. If the click_id already exists in the IdRegistry, the joiner assumes that the click has already been joined. If the joiner is able to register click_id into the IdRegistry, the joiner stores information from the query in the click and writes the event to the joined click logs.

Note that the above algorithm is susceptible to losses if the joiner successfully registers the click_id to the IdRegistry but fails to write to output logs. Later, we present simple techniques to minimize the loss in Section 3.3.2, and also discuss procedures to recover missing output in Section 3.3.3.

Figure 6 depicts the same Photon pipeline running independently in multiple datacenters, with the global state maintained by the IdRegistry. The rest of this section describes in detail the basic architecture, the design of each component and the technical challenges involved.

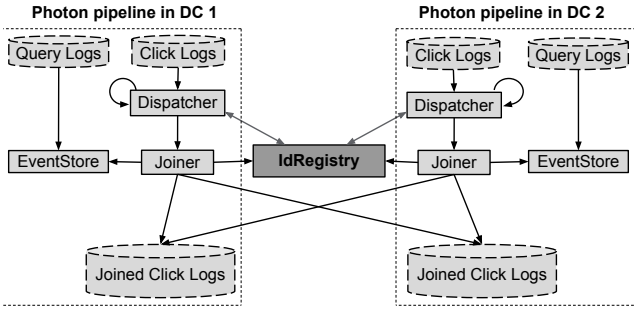

Figure 6: Components of pipelines running independently in two datacenters with the global state in the IdRegistry

3.1 Unique Event_Id Generation

Given the critical nature of the data they deal with, servers write all events (e.g. query and click) to a persistent storage like GFS [14] and also replicate them to multiple logs datacenters. When an event is recorded by a server, it is given a globally unique identifier (such as query_id or click_id as discussed before).

Production deployments consist of thousands of servers across the world that log all such events, a scale at which quick generation of event_ids is critical for a fast response to user actions. Independent generation of event_ids is a prerequisite to satisfy this requirement.

An event_id consists of three fields: *ServerIP*, *ProcessID* and *Timestamp*. These fields uniquely identify the server and the process on that server that generated an event, as well as the time the event was generated. Each server generates monotonically increasing timestamps for every event based on the current time. All the events in a log file are approximately sorted by their timestamp since multiple threads in the same process independently generate event_ids. This encoding of the location of an event, even if not entirely precise, plays a crucial role in finding the event in a large set of log files; this is discussed further in section 3.4.

Note that generating the timestamp for each event_id on each server locally may be adversely impacted by clock skew

on the local machine. To limit the skew to S seconds, we use the TrueTime API [8] that provides clock synchronization primitives. By using GPS and atomic clocks, the primitives guarantee an upper bound on the clock uncertainty. Each server generating event_ids executes a background thread that sends an RPC to one of the TrueTime servers every S seconds to synchronize the local clock.

3.2 Dispatcher: Ensuring At-Least-Once Semantics

The input to Photon consists of events that are written to log files. Those files continuously grow in size. Events can overflow to new files when log files reach a maximum size. At any given time, a Photon instance has to track several thousands of growing log files. Thus, the main objectives of the dispatcher are: monitor the growth of log files continuously, read new events in a scalable and timely way, and dispatch them to the joiner with minimal latency.

The dispatcher periodically scans the directory where log files are located to identify new files and check the growth of existing files. The dispatcher stores a per-file state in the local GFS cell. This state includes the files encountered by the dispatcher, and the next byte offset to read from each file. To achieve a high degree of scalability, the dispatcher uses many worker processes to concurrently process the log files. With the persistent state, the processing information is preserved across restarts of workers. All the concurrent workers share the persistent file state locally so that they can work on different parts of log files without stepping onto each other.

Before sending events to the joiner, the dispatcher looks up each event_id in IdRegistry to make sure the event has not been joined. This optimization technique significantly improved performance as observed from the measured results (Section 4).

3.2.1 Retry Logic

Note that a click can only be joined if the corresponding query event is available in the logs datacenter. As mentioned in Section 1, it is not uncommon for a subset of query logs to be delayed relative to the corresponding click logs. In such cases, a joiner will not be able to process click events until corresponding query events have arrived in the logs datacenter.

After reading a click event from a log file, the dispatcher sends an RPC to the joiner, and asynchronously waits for the response. When the dispatcher fails to receive a successful response from the joiner for a click, it saves the click in local GFS and retries it later. This guarantees *at-least-once* semantics for every event with minimum losses. To limit the rate of retries in case of repeated failures (e.g. due to a network congestion), the dispatcher employs an exponential backoff algorithm. If the joiner still fails to join a click after a certain number of retries, and the click is older than a threshold, the joiner will mark the click as *unjoinable* and return success to the dispatcher. In practice, unjoinable events represent a very small fraction of all the events.

3.2.2 Handling Datacenter Outages

The file state maintained by the dispatcher is local to the datacenter where it runs. If the local GFS datacenter suffers an outage, the dispatcher will be stalled. This does not impact the end-to-end availability of the Photon system because we have at least two copies of the Photon pipeline in different datacenters each of which continues processing independent of the other. Once the datacenter that suffered the outage recovers to a healthy state, if the persistent state is intact, the dispatcher can resume where it left off and starts by processing the backlog. Since majority of backlog events would have been processed by the pipeline in the other datacenter, these events will already exist in the IdRegistry, and the dispatcher will catch-up quickly from the backlog to start processing recent events. We use appropriate throttling of outstanding requests to the IdRegistry to avoid overloading it, and ensure adequate performance during catch-up.

In rare cases where the dispatcher file state cannot be recovered after a datacenter outage, we could manually initialize it to ignore the backlog and immediately start processing latest events from the current time, assuming the pipeline in the other datacenter has been operating without interruptions.

3.3 Joiner

The joiner is implemented as a stateless RPC server that accepts requests from the dispatcher and coordinates with the EventStore and the IdRegistry to perform the join as well as any specific business logic.

3.3.1 Processing Logic

After accepting an incoming RPC from the dispatcher, the joiner extracts the click_id and query_id from the click event. Then the joiner sends an asynchronous RPC to EventStore to lookup the query for the corresponding query_id. If the EventStore cannot find the query, the joiner returns an error code to the dispatcher, so that the dispatcher can retry the request after some backoff time. When a joiner receives an event from the dispatcher, it first checks if there are already too many requests in flight. If so, the request will be rejected and the dispatcher will have to retry it later. Such throttling is essential to maintain the smooth flow of processing of events through the joiner.

A successful lookup to the EventStore returns the query corresponding to the click. As the next step in processing, the query and the click are passed to a library called the *adapter*. Usually, the adapter simply combines the two events into one joined event and passes the result back to the joiner. However, it may also apply application-specific business logic for filtering, or it can force the joiner to skip the click based on certain properties of the query. Isolating such business logic to the adapter adds flexibility to Photon, so that any two realtime data streams can be joined without modifying the core infrastructure.

After receiving the joined click event from the adapter library, the joiner will attempt to register the click_id into the IdRegistry using an asynchronous RPC. If successful, the joiner will then proceed to append the joined click to the output log, stored in the logs datacenters. Since the IdRegistry guarantees at-most-once semantics, for any given click_id at most one joiner will be able to successfully register it, preventing duplicate joined clicks to occur in the output logs.

If the joiner is unable to successfully register the click_id into the IdRegistry, it will drop the event. We consider this as a *wasted join*. In our production deployment, since we have dispatchers running in at least two datacenters, Photon

reads each click at least twice. However, we minimize the wasted joins in the system by having the dispatcher do a lookup in the IdRegistry before sending the click to a joiner, and dropping the event if it is already present.

In contrast to the dispatcher, the joiner does not use a persistent storage to maintain any state. In addition, all joiners are functionally equivalent so we can use RPC-based load balancing to uniformly distribute the input requests from the dispatchers amongst all the joiners.

3.3.2 Minimizing Joiner Losses

The joiner writes an event to the output log only after successfully writing the event_id to the IdRegistry. Once an event_id is registered into the IdRegistry, another request to register the same event_id will fail. The IdRegistry guarantees this property. However, the lack of transactional atomicity between writing the output and registering the event_id can lead to problems as illustrated below while processing click events.

The problem arises when the IdRegistry successfully registers the click_id but the joiner times out or the RPC response to the register request gets lost in the network. Subsequent register requests from the same joiner will fail due to the existence of the click_id and the click will never be joined as a result. In our early test deployment, we observed a noticeable number of clicks missing from the joined output logs due to this reason.

To solve this problem, when a joiner commits a click_id to the IdRegistry, it also sends a globally unique *token* to the IdRegistry (consisting of the joiner server address, the joiner process identifier, and a timestamp) along with the click_id. The IdRegistry stores this information about the joiner as the value associated with the click_id. If the joiner does not receive response from IdRegistry within a certain timeout interval, it retries the request with the exact same token.

When the IdRegistry receives a request to register an existing click_id, it checks if the value for the click_id stored inside IdRegistry matches with the token in the request. If they match, it means the request is a retry from the same joiner that registered this click_id last time but did not receive the RPC response. In such a case, the IdRegistry returns *success* to the joiner, which will be able to output the joined click. This mechanism enables us to gracefully handle joiner retries and greatly reduce the number of missing clicks. In our production deployment of Photon, this technique reduced the number of missing clicks in joined output logs by two orders of magnitude.

Another potential source of event loss is due to unexpected crash of the joiner workers. Once a joiner sends an RPC to the IdRegistry, the latter may successfully register it into the IdRegistry but the joiner may crash and restart before receiving the acknowledgment and writing the joined click to output logs. Since Photon workers run on commodity hardware, such intermittent failures are inevitable, especially on large scale. After such a missed write, no other joiner can commit the event to output logs because the token kept in the IdRegistry does not belong to any joiner. We minimize the loss by enforcing an upper limit on the number of outstanding RPCs from any joiner to the IdRegistry. Once the joiner reaches the limit, it throttles the incoming requests from the dispatchers.

3.3.3 Verification and Recovery

As discussed above, Photon can miss events in the output joined logs due to absence of transactional atomicity between recording the event in the IdRegistry and recording in the output joined logs. Example scenarios that can surface this situation are: (a) the joiner crashes or restarts after writing to the IdRegistry, (b) logs storage loses data, (c) a bug in the code manifests itself after committing to the IdRegistry.

In normal conditions, observed losses represent a very small fraction – less than 0.0001 percent – of the total volume of joinable events. Consequently, the recovery scheme presented below is more useful to act as an insurance for rare cases such as bugs or catastrophic, uncontrollable failures.

Photon provides a verification system which takes each event in the input and checks its presence in the output. If an event in the IdRegistry is not present in the output log, we read the server address and process identifier from the token in the IdRegistry corresponding to the missing event, and identify the joiner. If the joiner had crashed or restarted, the event can be safely reprocessed without causing duplicates. To recover the event, we delete the missing event from the IdRegistry, and re-inject the event back into the dispatcher.

The recovery system only needs to read the tokens of events missing in the output logs. Hence, we can optimize storage cost of the IdRegistry by removing tokens of all joined events that have been successfully committed to the output logs. We scan the output joined logs continuously and clear the corresponding values from the IdRegistry. Thus the IdRegistry will only store tokens for lost events over a longer period of time, until they are recovered.

3.4 EventStore

EventStore is a service that takes a query_id as input and returns the corresponding query event. One simple approach to implement the lookup is to sequentially read the query logs and store the mapping from query_id to the log file name and byte offset where the query event is located. The mapping can be saved in a distributed hash table (e.g. BigTable [6]).

We built two implementations of EventStore based on the characteristics of our data: *CacheEventStore*, that exploits the temporal locality of events (for example, most of the ads are clicked shortly after the search query), and *LogsEventStore*, that exploits the fact that queries in a log file are approximately sorted by the timestamp of the events (see Section 3.1 for details).

3.4.1 CacheEventStore

CacheEventStore is an in-memory cache which provides high throughput and low-latency lookups for recent queries. It is a distributed key-value store similar to Memcached [12], where the key is the query_id and the value is the full query event. CacheEventStore is sharded by the hash of query_id and we use consistent hashing [17] to support smooth transitions during resharding. CacheEventStore is populated by a simple reader process that reads query logs sequentially and stores them in memory. CacheEventStore can keep several minutes worth of query events in memory. When it is full, the least recently used query events are evicted. Many of the events are never accessed because the number of foreign events is much smaller than the primary events stored in CacheEventStore. CacheEventStore is a best-effort system:

in case of a cache miss, the query lookup is forwarded to the LogsEventStore. It does not employ any kind of persistence mechanism and is there purely to optimize latency of primary event lookups, and save disk seeks by avoiding requests to the LogsEventStore: even though in order to populate it we end up reading all of primary logs, it is all in sequential reads and we avoid expensive random file access required by the LogsEventStore.

3.4.2 LogsEventStore

LogsEventStore serves requests that are not satisfied by the CacheEventStore. Normally such requests constitute only a small fraction (about 10%) of all lookups. But LogsEventStore may need to serve 100% of lookups during catch-up to process old logs, or when the CacheEventStore is having problems.

LogsEventStore serves requests by consulting a persistent *log file map* to determine the approximate position of an event in the primary log file. From that position, it reads a small amount of data sequentially to find the exact event. Log file map is a collection of entries that map an event_id to a particular offset within a log file where that event is located. These entries are emitted at regular intervals (every W seconds or M bytes) by the event reader as it reads primary log files to populate the CacheEventStore. Entries are stored in BigTable [6], which implements a distributed ordered map that allows efficient key range scans. To find the log file name and offset that is close to the event, the LogsEventStore scans a range of the file map BigTable key space beginning with *ServerIP:ProcessId:(Timestamp-W)* and ending just after *ServerIP:ProcessId:Timestamp*. It is not necessary for either of the keys to exist in the table, but as long as the primary event reader has been configured to emit map entries every W seconds, the result will contain at least one map entry which will put the reader within M bytes of the target event. By adjusting M and W we find the desired balance between the map table size and the amount of I/O required for each lookup.

4. PERFORMANCE RESULTS

Photon has been deployed in production for more than one year, and it has proven to be significantly more reliable than our previous singly-homed system. During this period, Photon has survived multiple planned and unplanned outages of datacenters without impacting the end-to-end latency. The physical placement of pipelines and the IdRegistry replicas is as follows:

- The IdRegistry replicas are deployed in five datacenters located in three geographical regions across the United States. These geographical regions are up to 100ms apart in round-trip latency. Each region has at most two IdRegistry replicas. Since we configured the IdRegistry to commit as long as three out of five replicas reach consensus, this setup ensures that we can cope with the complete outage of one region (as in case of a major natural disaster) without causing disturbance to production.
- All the other components in the pipelines (including dispatchers, joiners, etc.) are deployed in two geographically distant regions on the east and west coast of the United States. These two regions also have close physical proximity with logs datacenters.

The following statistics highlight the scale and efficiency of Photon:

- Photon produces billions of joined events per day. During the peak periods, Photon can scale up to millions of events per minute.
- Each day, Photon consumes terabytes of foreign logs (e.g. clicks), and tens of terabytes of primary logs (e.g. queries).
- More than a thousand IdRegistry shards run in each datacenter.
- Each datacenter employs thousands of dispatchers and joiners, and hundreds of CacheEventStore and LogsEventStore workers. These components are replicated in two or more datacenters globally.

To satisfy the mission-critical nature of the data, Photon pipelines monitor a wide variety of performance metrics. Below, we present some performance numbers from our real-world deployment. Note that the time range and granularity in some of the following graphs are not specified for business confidentiality reasons.

4.1 End-to-end Latency

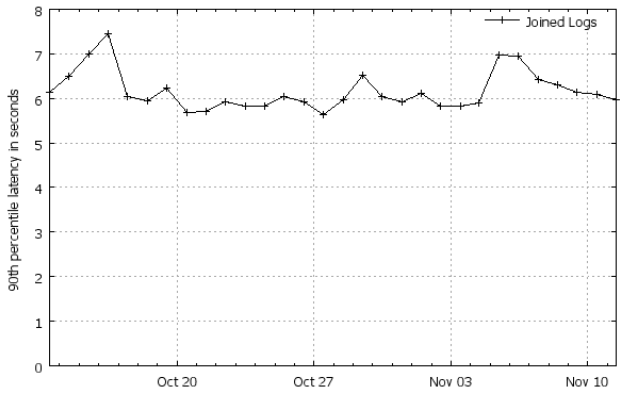

Figure 7: End-to-end latency of Photon

We define the end-to-end latency of each joined event as follows: when the joiner is ready to output a joined event, we export its latency as the current time minus the time when the event was logged — which is obtained from the event id's timestamp. The end-to-end 90th-percentile latency of Photon is computed based on the latency of each of the joined events.

Figure 7 plots the end-to-end 90th-percentile joining latency over a 30-day period. This graph shows that 90% of the events were joined in less than 7 seconds. This low latency is a direct result of the fact that the multiple stages in the pipeline communicate with each other through RPCs without the need to write intermediate data to disk storage (see Section 3).

One potential performance bottleneck arises from the time spent in disk seeks by LogsEventStore. With in-memory caching (i.e., CacheEventStore), Photon reduces the number of disk-based lookups significantly. Figure 8 shows the cache hit ratio for the CacheEventStore. The graph shows that most lookups are successfully handled by CacheEventStore

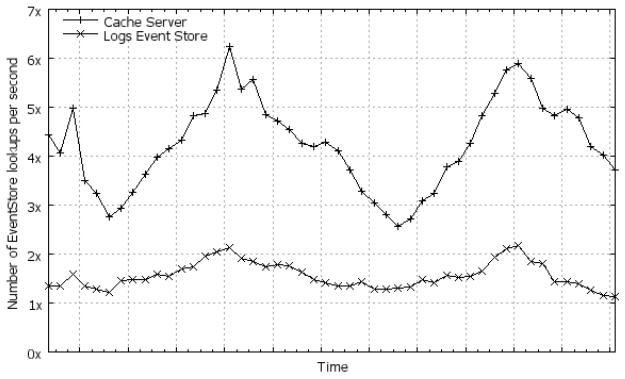

Figure 8: Photon EventStore lookups in a single datacenter

and only a small volume of traffic is processed by the slower LogsEventStore. Note that the volume of traffic to LogsEventStore stays relatively flat compared to the traffic growth - this means that we get even higher cache hit rate when the traffic increases.

4.2 Effect of Datacenter Outages

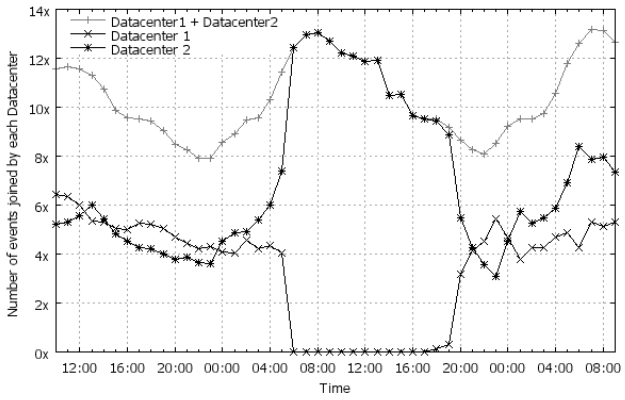

Figure 9: Photon withstanding a real datacenter disaster

Figure 9 shows the numbers of joined events produced by production pipelines in two separate datacenters over a period of time. When both datacenters are healthy, each processes half of the total events. However, when one of the datacenters suffers from a disaster, the other datacenter automatically starts handling the complete traffic without any manual intervention.

Similarly, when one of the IdRegistry datacenters goes down, the end-to-end performance of Photon is not impacted: since we are running five PaxosDB replicas in our setup, it lets us tolerate up to two datacenter-level outages.

4.3 IdRegistry Performance

In Section 2.2, we described how the IdRegistry batches multiple client requests into a single PaxosDB transaction to increase the throughput. Figure 10 shows the effectiveness of server-side batching at a single IdRegistry shard. This particular shard receives hundreds of client write requests per second. However, after batching, the number of PaxosDB transactions is only 6 to 12 per second.

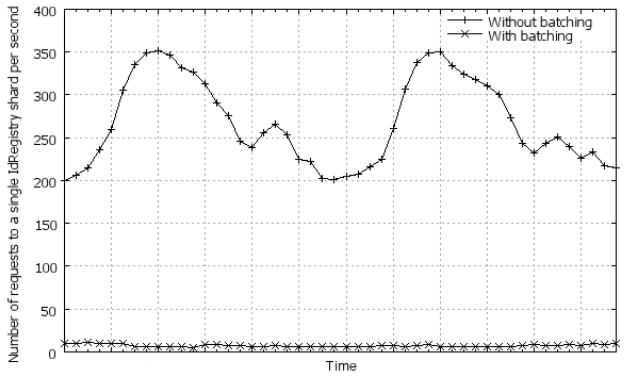

Figure 10: Effectiveness of server-side batching in the IdRegistry

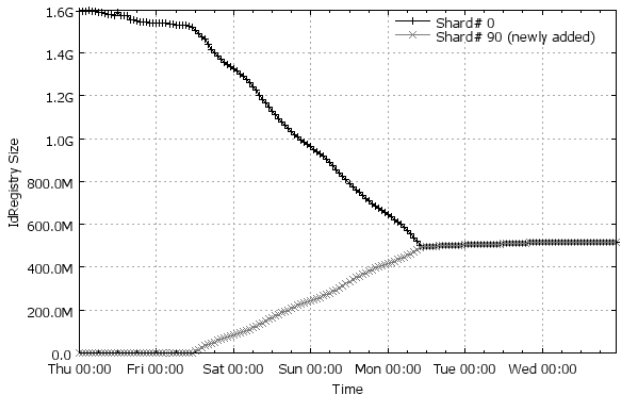

Figure 11: IdRegistry dynamic time-based upsharding

Figure 11 shows the effect of the IdRegistry dynamic resharding based on timestamp. Before upsharding, the size (= number of keys * bytes per key) of shard 0 was around 1.6GB. We tripled the number of shards and made it effective starting at Fri 12:00. This caused the size of shard 0 to drop and the size of newly added shard 90 to increase, until they both converged to 500MB. The IdRegistry garbage-collects event_ids that are older than 3 days, which explains why the old and new shards converge to the same size after 3 days.

4.4 Resource Efficiency

As mentioned in Section 2, we are able to tolerate datacenter-level failure by running independent pipelines in two datacenters, which clearly leads to some redundancy in the performed work. Recall from Section 3.2 that the dispatcher uses IdRegistry lookups to reduce the number of wasted joins.

Figure 12 shows that even though we read all events in both the datacenters, the optimization works very well in practice: less than 5% of the events are processed by both joiners.

This lookup technique also reduces the dispatcher catch-up time required after an extended downtime.

Figure 13 shows the dispatcher in one datacenter coming back after a 2-day downtime, and benefiting from the optimization. As soon as it was brought up, the dispatcher

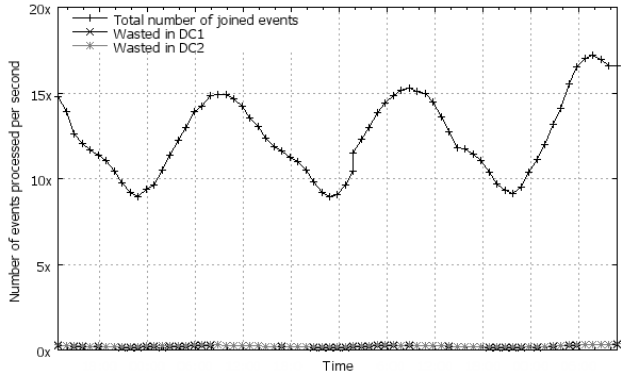

Figure 12: Wasted joins minimized by the dispatcher performing an IdRegistry lookup before sending events to the joiner

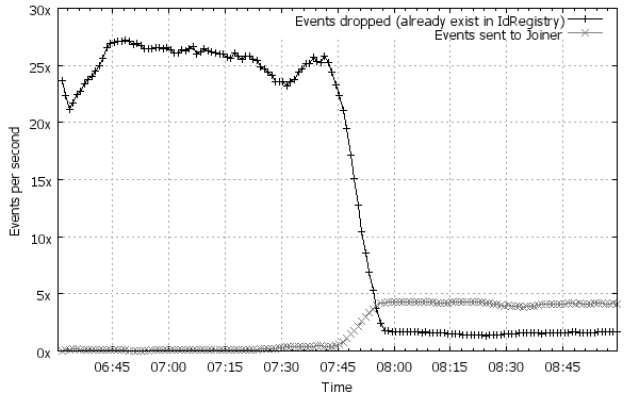

Figure 13: Dispatcher in one datacenter catching up after downtime

started to reprocess all the events of the last two days. Because a majority of these events had already been processed by the other healthy datacenter, the dispatcher caught up very quickly from the backlog.

5. DESIGN LESSONS

While designing and implementing many versions of the joining system, several design principles and lessons have emerged. Our experience is summarized below.

- To build a scalable distributed system based on Paxos, we need to minimize the critical state managed by the system, and ideally restrict it to meta-data only. Writing data through Paxos is very expensive because the data needs to be replicated to multiple datacenters across wide-area network in a synchronous fashion. Note that if the size of data is relatively small, it may be feasible to store the complete data (instead of meta-data) in Paxos as described in Spanner [8].
- In Photon, IdRegistry is the central point of communication among multiple pipelines maintaining consistent global state, and could turn into a bottleneck that limits scalability. We ensured that dynamic resharding of a running system is a first-class citizen in the design to avoid this limitation.
- RPC communication among workers helps in reducing the end-to-end latency, compared to using disk as the means of data communication. But using RPC requires application-level check for lost RPCs and retries to satisfy reliability guarantees. Special care should be exercised in throttling asynchronous RPCs to avoid overloading various servers such as the IdRegistry. Designating a single job (in this case, Dispatcher) to be responsible for retries facilitated the implementation of all other jobs to communicate via RPCs.
- It is better to isolate the specialized work to a separate pool of workers in order to make the system more scalable (in other words, divide and conquer). In our previous joining system, the work of the EventStore and the joiner was performed by the same component, and we did not have a CacheEventStore. Since LogsEventStore requires opening files and doing disk seeks, workers had affinity to only process the clicks where the corresponding query may be in the same physical file. This limited the scalability of the system. For a given category of work, designing each worker to be capable of operating on any particular piece of input makes the system more scalable.
- Some stream processing systems [26] recommend grouping individual events into a bigger batch. This works well only if all the events in a batch can be processed at the same time. For our case, it is very common to see a click event before its corresponding query event. If we group multiple clicks into a batch, then the whole batch cannot be committed until the corresponding queries are available. Hence, system designers need to make a careful decision on whether to batch or not.

6. RELATED WORK

There is a rich body of work on join algorithms in parallel and distributed RDBMS [19]. Over the last decade, researchers have also proposed techniques to join multiple streams continuously [1, 3, 5, 13, 15, 20, 24–26]. To the best of our knowledge, none of the existing literature addresses this problem under the following system constraints - eventually exactly-once semantics, fault-tolerance at datacenter-level, high scalability, low latency, unordered streams, delayed primary streams. Teubner et al. [24] used modern multi-core systems as opposed to commodity hardware. Blanan et al. [3] used MapReduce [10] to join streams, which is more suitable for a batch system rather than a continuously running one. Rao et al. [21] proposed using Paxos to maintain consistent database replication while being fault-tolerant. Unlike Photon, they store all Paxos replicas within a single datacenter, and hence are not impervious to datacenter disasters. Zaharia et al. [26] proposed breaking continuous streams into discrete units, which is not acceptable for Photon since we may see some clicks in a discrete stream before their corresponding query has arrived at the logs datacenter. Das et al. [9] proposed approximate joins as opposed to exact ones. Most of the existing join algorithms also assume that events in both input streams are sorted by the shared identifier, hence, they operate within a window to join [16]. In Photon, click can be arbitrarily delayed relative to its query.

Spanner [8], Megastore [2] and DynamoDB [11] provide consistent replication across datacenters as a storage service.

Both Spanner and Megastore do not currently allow applications to perform server-side batching of multiple client requests into a single Paxos commit.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we described our experience in building and deploying a distributed stateful stream joining system that can withstand datacenter-level outages. The dispatcher keeps sending an event to the joiner until the IdRegistry server registers the event, ensuring at-least-once semantics. The IdRegistry is the heart of the system, as it ensures that no event is present in the output more than once. The dispatcher and the IdRegistry together guarantee that the only case where an event is lost is when the system commits the event to the IdRegistry, but fails to write the joined event to output. We presented an off-line mechanism to recover such unlikely losses. There is no single point of failure in the system, and all the components scale linearly with increased volume of traffic, including the scaling of critical state through resharding of IdRegistry. Asynchronous RPC-based communication among components reduces the latency, and helps achieve the goal of timely availability of rich statistical data.

We are working to extend Photon to join multiple continuous streams and scale sub-linearly. To reduce the end-to-end latency even further, we plan to have another fast, best-effort RPC path where the servers that generate query and click events directly send RPCs to the joiners (instead of the dispatcher waiting for these events to be copied to the logs datacenter). This will allow majority of events to be processed by the fast path, leaving the current logs-based system to process the missing events.

References

[1] D. J. Abadi et al. "The Design of the Borealis Stream Processing Engine". *Proc. of CIDR* 2005, pp.277-289.

[2] J. Baker et al. "Megastore: Providing scalable, highly available storage for interactive devices". *Proc. of CIDR* 2011, pp.223-234.

[3] S. Blanas et al. "A comparison of join algorithms for log processing in Mapreduce". *Proc. of SIGMOD* 2010, pp.975-986.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. "Paxos made live: an engineering perspective". *Proc. of ACM PODC* 2007, pp.398-407.

[5] S. Chandrasekaran and M. J. Franklin. "Streaming queries over streaming data". *Proc. of VLDB* 2002, pp.203-214.

[6] F. Chang et al. "Bigtable: A Distributed Storage System for Structured Data". *ACM TOCS* 2008, 26.2, pp.4:1-4:26.

[7] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM* 13 (6): p377-387, 1970.

[8] J. C. Corbett et al. "Spanner: Google's Globally-Distributed Database". *Proc. of OSDI* 2012.

[9] A. Das, J. Gehrke, and M. Riedewald. "Approximate join processing over data streams". *Proc. of SIGMOD* 2003, pp.40-51.

[10] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters". *Proc. of OSDI* 2004, pp.137-149.

[11] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". *Proc. of SOSP*. 2007, pp. 205-220.

[12] B. Fitzpatrick. "Distributed Caching with Memcached". Linux Journal, Issue 124, 2004, pp.5.

[13] B. Gedik, P. S. Yu, and R. R. Bordawekar. "Executing stream joins on the cell processor", *Proc. of VLDB* 2007, pp.363-374.

[14] S. Ghemawat, H. Gobioff, and S-T Leung. "The Google File System". *19th Symposium on Operating Systems Principles* 2003, pp.20-43.

[15] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. "Joining multiple data streams with window constraints". *Computer Science Technical Reports*, #02-115.

[16] J. Kang, J. F. Naughton, and S. D. Viglas. "Evaluating window joins over unbounded streams". *Proc. of VLDB* 2002, pp.341-352.

[17] D. Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". *Proc. of ACM SOTC* 1997, pp.654-663.

[18] L. Lamport. "The part-time parliament", *ACM TOCS* 16.2 1998, pp.133-169.

[19] P. Mishra and M. H. Eich. "Join processing in relational databases". *ACM Computing Surveys* 1992, 24(1), pp.63-113.

[20] L. Neumeyer et al. "S4: Distributed Stream Computing Platform". *Proc. of KDCloud* 2010.

[21] J. Rao, E. J. Shekita, and S. Tata. "Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore". *Proc. of VLDB* 2011, pp.243-254.

[22] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial", *ACM Computing Surveys* 22 1990, pp.299-319 (1990).

[23] D. Shasha and P. Bonnet. "Database Tuning: Principles, Experiments, and Troubleshooting Techniques". *Proc. of SIGMOD* 2004, pp.115-116.

[24] J. Teubner and R. Mueller. "How soccer players would do stream joins". *Proc. of SIGMOD* 2011, pp.625-636.

[25] J. Xie and J. Yang. "A survey of join processing in data streams". *Data Streams - Models and Algorithms* 2007, pp.209-236.

[26] M. Zaharia et al. "Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters". *Proc. of HotCloud* 2012, pp.10-10.