Cost-Efficient and Reliable Reporting of Highly Bursty Video Game Crash Data

Drew Zagieboylo
Electronic Arts
dzagieboylo@ea.com

Kazi A. Zaman
Electronic Arts
kzaman@ea.com

ABSTRACT

Video game crash events are characterized primarily by large media payloads and by highly bursty traffic patterns, with hundreds of thousands or millions of reports being issued in only a few minutes. These events are invaluable in quickly responding to game breaking issues that directly impact user experience. Even the slightest delay in capturing, processing and reporting these events can lead to user abandonment and significant financial cost.

A traditional standalone RESTful service, backed by a vertically scaled SQL database is neither a reliable nor cost-effective solution to this problem. An architecture that decouples capture and persistence and uses a horizontally scalable NoSQL database is not only easier to provision, but also uses fewer cpu and memory resources to provide the same end to end latency and throughput.

By replacing our RESTful implementation with one that takes advantage both of the aforementioned design and multi-tenant provisioning, we have reduced our dedicated cpu footprint by 63% and memory footprint by 59%. Additionally, we have decreased our data loss during spikes to essentially 0, maintained sub-second persistence latency and improved query latency in the average case by 54% with only a 3% sacrifice for worst case queries.

CCS Concepts

•**Computer systems organization** → **Cloud computing;** *Real-time system architecture;* Reliability; Maintainability and maintenance; •**Applied computing** → Media arts;

Keywords

Cloud Infrastructure, NoSQL, Reliability, Crash Reporting, Cost Efficiency

ICPE'17, April 22 - 26, 2017, L'Aquila, Italy

ISBN 978-1-4503-4404-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3030207.3044529

1. INTRODUCTION

Crash reporting is an invaluable system in the world of online gaming. As any software developer knows, the best tested code is not without bugs and even in the case where the code "works as expected" problems will arise due to the limitations of software and hardware infrastructure. Most games will experience their highest peak simultaneous users upon game or expansion release; this implies that any given game will likely be under its most demanding load when it has had the least amount of exposure to users (and therefore realistic testing). In such situations where bugs are not resolved expediently, there can be significant user backlash and abandonment [13]. Thus the crash reporting system (CRS) needs to be highly available and reliable; dropped crash reports and high query latencies limit the data available to developers and increase time to mitigation.

However, there are several challenges to implementing a CRS. One such challenge, which is common among many software applications, is the minimization of client overhead. Game clients cannot wait many seconds or minutes to submit crash reports, since submission time directly increases recovery time and user experience [14]. If a CRS is unresponsive due to the numerous clients trying to report simultaneously, the client is forced to disconnect and fails to submit the crash data or must submit at a significantly later date.

Another practical restriction is cost and scalability. As previously noted, many major problems are encountered at times with high peak simultaneous users, and often affect the vast majority of the current users; this in turn leads to an extremely bursty traffic pattern. The classic tradeoff between over and under provisioning (a.k.a. the tradeoff between cost and negative impact, respectively) applies quite starkly to this use case and needs to be closely considered. Lastly, crash reports that occur near simultaneously are often indistinguishable except for detailed stack traces and memory dumps. This makes keyed partitioning effectively useless for balancing resource usage during peak load. In such architectures, horizontal scaling of hardware resources can be effectively nullified and provisioning for peak load means adding memory and/or I/O bandwidth to a single machine.

In this paper, we describe how the New CRS is an improvement over the simplistic model of RESTful service for crash reporting. We address all of the aforementioned challenges, while comparing and contrasting other solutions that we considered. Our goals are specifically to reduce data loss, increase cost-efficiency (as measured by report throughput

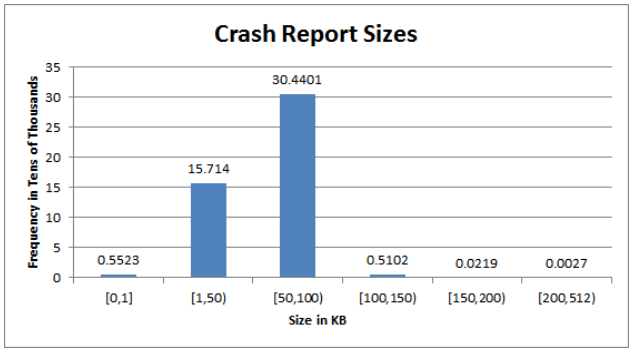

Figure 1: Sample of Reports from AAA Title Beta

per cpu and memory provisioned) and maintain or improve query latency. This paper will describe each of the architectural modifications and technology choices that we made in order to achieve this goal. For each component in the design, this paper will additionally detail both the performance benchmarks we ran during evaluation and the behavior of the sytem since its release to our production environment. Once evaluating the results of our efforts, this paper will explore further work that can be done to improve usability of the system, while still maintaining high throughput, reliability and scalability.

2. BACKGROUND

2.1 Crash Report Taxonomy

EA crash reports have a small number of required fields that help users to differentiate reports based on game, crash type, etc. Many game teams also opt to send detailed debugging data that notably includes any or all of the following: stack trace, screenshot, and heap memory dump. These blob fields represent the bulk of the data volume and are encoded in base64. For the purposes of this paper we will concern ourselves primarily with the following fields:

- *sku* - the game/game-platform combination on which the crash originated
- *submitTime* - the time at which the crash report became readable through the CRS
- *reportId* - an internal UUID assigned to each crash report by the CRS

In addition to crash reports, the CRS also accepts 'session' events, which simply represent a session beginning either for a game client or game server. These are always small since they contain the same required fields as reports but no media or large text fields.

2.2 Crash Report Workload Profile

Crash reports in the system have an approximately bimodal distribution of 'large' and 'small' reports. Large reports are those that include the media or blob text fields listed above, these range anywhere from 10 KB to 512KB. Small reports contain no media or large text fields (except for a stack trace) and range from several hundred bytes to about 5KB. The average size of all reports is 5KB, but that can be a misleading measurement due to the bimodal distribution mentioned above. For example Figure 1 represents the sizes of crash reports from a single EA game during its open beta; for this game the average report size is significantly higher, around 100 KB. The design and benchmarks listed later in this paper reflect this 100KB crash report profile in order to not only prepare for the worst case, but also to better future proof for the time when more game teams decide to provide large media in their crash reports.

The baseline submission load for the CRS is between 350 and 800 requests per second, depending on the time of day. Of those requests, approximately between 30 and 50 are crash reports and the remainder are session events. Based on the above average data size, we can extrapolate that our baseline data throughput for crashes is 250KB/s. Peaks approach 2500-3000 crash reports per second and are short lived. Figure 2 shows incoming report events per minute during the open beta for a AAA title. This depicts the unique characterization of large but short-lasting spikes of crash traffic.

Lastly, the CRS maintains crash report data for 90 days. The majority of this data is not queried with any frequency, as game developers are usually concerned with only the most recent crashes; nevertheless old data is retained to allow post mortems and other long term analysis. The primary implication of this fact is that our database needs to store approximately 2TB of data (excluding replication), while only a small fraction of that needs to be quickly accessible for fast read or update operations.

2.3 Crash Report Query Profile

On average there are approximately 10 concurrent queries per second. 99.0% of queries return fewer than 2000 crash reports and 99.9% of queries return fewer than 20000; the vast majority of queries return fewer than 100 reports. Again, due to the bimodal distribution of report sizes, this causes a vast range in query size from hundreds of KB to hundreds of MB. This paper will further discuss performance characteristics in the Benchmarking section.

2.4 Hardware Resources

We used Amazon Web Services' Elastic Cloud Computing to provision our hardware resources for this implementation. AWS EC2 allows us to add or remove hardware easily with various configurations. For this project we used m3.xlarge instances, running CentOS 6. M3.xlarge instances have 4 virtual cpus, 15 GB of RAM, approximately 100 MB/s of network bandwidth and 80 GB of ssd storage. For componenents that required additional storage, we attached Amazon Electronic Block Storage, which provided persistent virtual ssd drives [3].

M3.xlarge instances provide the maximum network bandwith per dollar of all AWS instance types. Flux7 has performed Amazon-recommended benchmarking with i-series I/O optimized instances and m3 instances; while i-series instances do provide higher bandwidth per instance, the dollar cost per MB/s of bandwidth per hour is $0.005456 for i2.8xlarge and $0.00266 for m3.xlarge on-demand instances [1]. No cheaper instance provides better bandwidth than the m3.xlarge, according to Amazon's network performance descriptions and these tests. In practice, we have also noticed a network cap of 100-130 MB/s for m3.xlarge instances, corroborating the findings of Flux7.

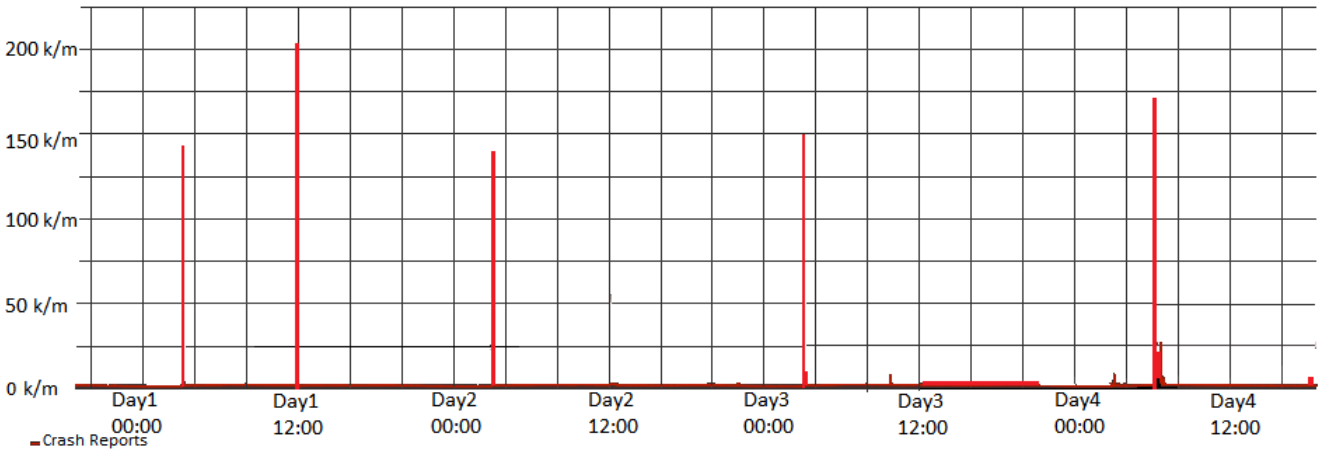

Figure 2: Incoming Crash Reports in Requests per Minute During Public Beta

3. PREVIOUS WORK

3.1 Other Research

While crash reporting has been researched previously, most work focuses on techniques for analyzing crash data. For example, there are many papers that describe processes for deduplicating crash reports based on memory and stack similarities [9, 14]. This paper seeks to fill a hole regarding infrastructural design and performance analysis. The CRS detailed in this paper provides the mechanism for information storage and retrieval, which is then used by downstream applications for bug detection.

3.2 Legacy CRS

In the introduction to this paper, we highlighted many of the challenges faced by a production scale crash reporting system. This section details how the Legacy CRS at EA handled some of those challenges and failed to account for others. The legacy implementation was a single-tiered capture and persistence service. Several servers, called collectors, in a company data center were load balanced behind a virtual IP to which clients could connect. After receiving client crash reports these collectors then wrote them into one of three MySQL databases. Other servers, called reporters, handled read requests; game developers used an API to connect to these servers to retrieve crash report data. Figure 3 depicts this design with game client connections on the left and game developer queries depicted on the right.

To avoid high client latencies in the event of a database outage or sluggishness, the collectors used an in-memory queue to buffer crash reports. Asynchronously, they were removed from the queue and then written to persistent storage. In order to increase write throughput and storage space, the application manually sharded data, based on crash report sku, to different MySQL database servers. Each MySQL server also had a matching slave server that was used by the reporters in order to prevent heavy read load from impacting write throughput.

3.2.1 Legacy CRS Shortcomings

While the above attempts to mitigate or solve performance and scale problems were improvements over a completely unintelligent RESTful service, they had some major flaws. First, the collectors maintained in-memory queues whose sizes were constrained by system memory; once the server ran out of heap space, the collector would crash and all reports in that queue were effectively discarded.

Next, there were also a few fundamental flaws with the data partitioning approach taken by the Legacy CRS. As previously mentioned, when a crash report spike occurs, it is usually due to a single game; this means that most reports will have the same one or two skus (one for each game platform that is affected). In this case, the data partitioning scheme listed here has no effect since utilization for one MySQL server will be maxed out, while the others lay mostly idle. Moreover there was no intelligent system for choosing how to map each sku to each database; two games that had higher than average crash rates could end up on the same server. This also lead to uneven MySQL utilization.

The Legacy CRS also had usability issues for game developers which arose due to the MySQL master-slave architecture. This design created an eventual consistency model for the reporters; data would be available for read at some undefined point after it had been written to the MySQL master (i.e. once the slave had replicated it). To make this situation more complicated, each MySQL master-slave pair had its own independent 'lag'; there was no central way to measure inconsistency. During a crash spike, slave lag would increase significantly; querying the last five minutes worth of crashes would certainly result in an incomplete list of crash reports since some number of them had not yet been replicated from master to slave. Due to the indeterminate nature of the consistency there was no way for the API to compensate for this 'pseudo-missing' data.

Finally, since this entire system was built at an in-house data center, there was no ability to launch new application or database servers during a crash spike. It would take days to coordinate hardware provisioning and setup, rather than the minutes or hours needed to either horizontally or vertically scale any piece of the system.

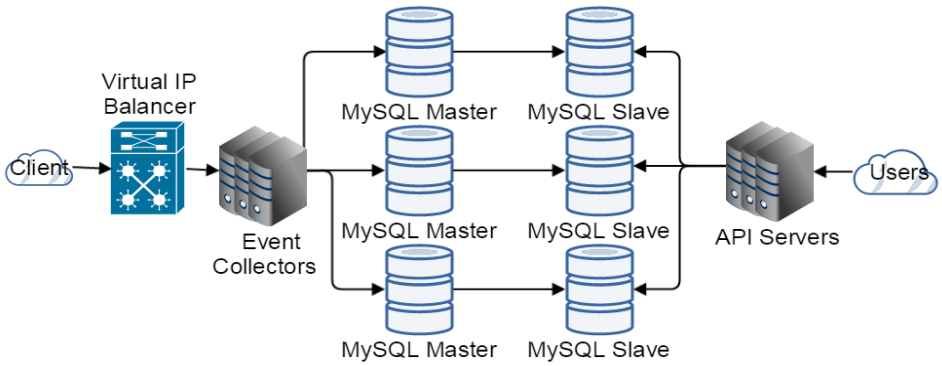

Figure 3: Legacy CRS Architecture

4. DESIGN OVERVIEW

4.1 Technologies

In this design we chose to use Kafka and Couchbase to implement the messaging and persistence layers of the design.

Kafka is a distributed message broker, which allows for highly scalable and redundant message publishing and consumption. Kafka performance is bounded primarily by network and disk I/O bandwidth; which is consumed via publishing, consumption and replication [4].

Couchbase is a horizontally scalable key-value store built on top of Memcached. The primary API is a simple GET-PUT interface, where the key is a string and the value is JSON document (although Couchbase does also support arbitrary binary data). Starting with Couchbase 4.0, it also offers document indexing and querying with an SQL called N1QL [8].

4.2 Persisting Crash Reports

When a game client or server experiences a crash, the following chain of events occurs:

1. Via HTTP, the client connects to a lightweight capture service and uploads the crash report.
2. While this connection is open, the capture service writes the report to Kafka and appends the report to a local file on disk.
3. Upon completion of these writes, the connection to the capture service is closed.
4. A consumer process reads the capture report from the Kafka queue.
5. The consumer persists the report to Couchbase.
6. The consumer updates a MySQL table that is used to indicate to the reporter that the crash data is available to be read.
7. The consumer updates a time-series counter (stored in MySQL) which tracks total sessions and crash incidents.[1]

[1]Step (7) is taken wholesale from the Legacy CRS; the MySQL-stored time series metrics were not a significant concern when developing the New CRS and so that piece of the system was left untouched. While improvements could certainly be made there as well, it is not the focus of this paper and will be largely undiscussed.

Notably, the legacy collector has been refactored into two processes: capturing data via the http service and Kafka; and persisting data via a Kafka consumer. Additionally, crash report storage has been migrated from MySQL to Couchbase, while the pre-existing MySQL has been converted into index storage.

4.3 Querying Crash Reports

While these changes were made in order to increase write throughput and reliability, they also incurred some changes to the reporter. Since the crash reports are no longer stored in an rdbms, there needs to be a mechanism for translating user query filters (such as time range and sku) into a list of document keys. MySQL is used as an 'index' to map submitTime, reportId, and sku to a given document key. This generates the following set of actions when querying the CRS:

1. User connects to reporter and supplies query parameters.
2. Reporter executes a MySQL query to retrieve the relevant document keys.
3. Reporter retrieves all of the reports from Couchbase in parallel.
4. Reporter applies non-indexed query filters, serializes the reports and returns them to the user.

This design is highlighted in Figure 4, and is visually separated into the three phases of Capture, Persistence and Reporting.

5. REPORT CAPTURE SERVICE

5.1 Overview

The capture service is a taxonomy-independent and pre-existing EA application which has exactly one job: accept incoming HTTP payloads and persist them with very high fault tolerance and minimal latency. Prior to work on the new CRS, this service was already in use for the purpose of receiving and storing game telemetry. It makes data available for downstream processing by storing it on local log files

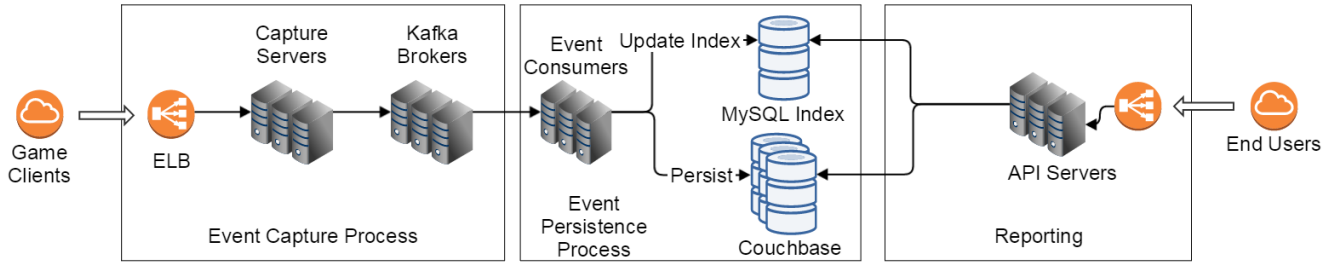

Figure 4: New CRS Architecture

and then rotating those files every 15 minutes to S3. From S3, scheduled ingestion processes pull the data into other locations in order to translate and process it.

The capture service runs on m3.xlarge EC2 instances and uses an Elastic Load Balancer to distribute work across servers.

Although this was a nearly ideal candidate for crash report capture, the batch processes involved in moving data to and from S3 had too much end to end latency. Users of the Legacy CRS expected near real time availability of crash reports, not multi-hour turn around times. To solve this, we replicated the legacy system of an in-memory queue by using a distributed messaging queue (i.e. Kafka). The capture service now also routes incoming data to a pre-configured Kafka topic based on the client URI; in this way crash reports are filtered out from other data streams and published to Kafka.

6. DATA PERSISTENCE

6.1 Crash Event Consumer

Once crash data has been stored safely in Kafka, it can at any time be consumed into a queryable data store. To accomplish this, we used the simple Kafka consumer api to build an application that reads the crash data stream, parses the events and writes them to Couchbase. This API allows the consumer to be a distributed application whose parallelism is limited primarily by the number of partitions in the Kafka topic from which it is reading. Adding or removing a new consumer instance will automatically rebalance the assignment of partitions to consumers, thus making hardware failure and horizontal scaling easy to handle. While the main purpose of the consumer is straightforward, there are a few practical considerations: how is the Couchbase document key generated; how are errors handled; and how does the MySQL index get updated.

6.1.1 *Key Generation*

The first and last points here are closely interrelated. In the Legacy CRS, each report was given a unique id by inserting a new row into a MySQL table with an auto-increment column; then the report was written to another table using that value as its reportId. This order of events creates data inconsistency; there is no order enforced on the stream of incoming reports and it becomes impossible to guarantee that a client has read all of the data that is actually present in the system. It follows that in the New CRS, the report is first written to Couchbase and then the MySQL index table is updated; however this means that the reportId is not generated until after the document is already stored in couchbase. Some other value must be used as a Couchbase key.

Here we take advantage of another feature of the capture service. It assigns a UUID to each event that it receives and attaches that in a header field of the published message. We have opted to use a hash of this UUID in order to uniquely generate each crash report key. This gives us the convenient side-effect that writing a report is idempotent; reprocessing or retrying will not generate undue duplicate entries and it does not require any extra logic to enforce. Importantly, we use a truncated hash of the UUID in order to reduce the memory overhead of storing a single report in Couchbase. The truncation to 16 bytes reduces the Couchbase key length by 20 bytes per document. Although it is not a necessary step, it is one that allows more data to be stored in memory in Couchbase and thus more data that can be queried at peak throughput.

6.1.2 *Data Partitioning in Couchbase*

As previously noted, the Legacy CRS had a hotspot issue caused by the difficulty of partitioning similar-looking crash reports. Couchbase partitions data randomly by hashing document keys, which leads to even usage across the cluster. Regardless of the key generation implementation, employing Couchbase as a storage engine removes the former hotspot bottlenecks.

6.1.3 *Index Format*

The MySQL index is represented as a single MySQL table with four columns, most of which are repeated from above:

- reportId - the ID used by clients to query a given report or range of reports
  - Primary key
- submitTime - the time at which the crash report became readable through the CRS
  - Partition key
- sku - the game/game-platform combination on which the crash originated
  - Indexed
- storageId - the couchbase document key

After writing a document to couchbase an entry is made into this table, where reportId is generated as an auto-increment value. This is the only bottleneck in the entire

pipeline which cannot be horizontally scaled; luckily, it is a very fast operation. The average latency for this step is < 1 ms. However, the critical section must be shorter than these observed latencies since we have supported more than 250 index entries per second. We have not measured precisely the length of this critical section; ultimately it will dictate an upper bound on consumer throughput at 1/len(critical section) reports per second.

For any user queries which apply other filters (e.g. errorCode), the reporter application must scan all of the documents that fall in the specified time range for that sku; much like querying via an unindexed field in traditional rdbms. The cost to index a new field would be to alter this table to include a new column, scan every crash report and update that entry in the index table. This is a highly expensive operation and a notable drawback of this implementation; this paper will address this concern in the future work section.

6.1.4 Error Handling

The consumer needs to handle several different error cases: malformatted data, temporary infrastructure errors, and non-recoverable errors.

The first is the easiest case to handle but is quite frequent, especially in testing. The consumer simply drops data that it cannot parse or that is is missing any required fields. The benefit of using Couchbase for storage is that we can actually persist the entire payload, even if it has unexpected fields or fields with unexpected types; this minimizes our 'required' field list to include only fields which need to be indexed for querying. However, there is a significant downside to handling user errors asynchronously; the New CRS cannot respond to the client with debug information about why a particular report was dropped. In our integration environment, we have enabled payload inspection on the capture service to allow client to receive detailed 4XX errors, if they so choose; however this drastically reduces the throughput of a single capture instance and is not cost-effective for a production environment.

Temporary errors occur frequently in production cloud (and physical data center) environments. They include but are not limited to: temporary network outages, latency spikes and short failover periods during server hardware failure. They are also fairly straightforward to handle by ensuring that only idempotent operations be retried and that offsets are committed to Kafka only once the corresponding data has been persisted. To this end, the consumer will stop reading data from a Kafka partition if it encounters a temporary error during processing. Then it will repeatedly retry until the error is resolved. Since all operations other than index updates in the persistence process are idempotent (and a successfull index update indicates successful persistence) it is safe to simply retry the entire procedure.

Non-recoverable errors should be rare and require manual intervention to resolve. For example, a server might run out of disk space or memory due to a faulty configuration or simple accident, causing an error. The consumer handles these cases identically to temporary errors because it can be difficult to distinguish between them. System administrators are notified of such errors via email when any of the consumer threads are stuck in the retry state for an extended period of time. This configurable period effectively dictates the minimum time to detection for any serious issues.

6.2 Alternative Storage Options

Before finally settling on Couchbase as our database we considered several other technologies. Ultimately Couchbase won out for a few reasons: it provides some controls for multi-tenancy; it has competitive write throughput per node; it requires no complicated logic to horizontally scale and it is simple to configure and maintain. While it is often overlooked in benchmarks and white papers, this last point is critical to creating a practical production system. A difficult to optimize database leads to higher costs in both performance and developer-hours spent debugging esoteric issues. This section will introduce the competitors that we considered and will conclude with comparisons to couchbase.

6.2.1 SQL (MySQL)

The'least-effort' implementation for us would have been to leave the legacy MySQL system in-place, potentially modifying the data schema to improve write performance. As we've already described there are significant issues with this approach; primarily the inability to easily remove hotspots and extra overhead required to shard data intelligently. Therefore, to use MySQL as our database we could either simply accept the occasionally high latency spikes and unavailability of data or we could attempt to improve the sharding algorithm. The former was deemed unacceptable for obvious reasons.

To implement the latter would have required maintaining a dynamic mapping for each report to shard mapping (we would have to remove the sku-based sharding to avoid hotspots), in the form of a hash function. Additionally we would have had to implement replication, cluster expansion and failover algorithms; at this point we would already be implementing most of the features provided by NoSQL storage engines. For almost any realistic use case, this is not worth the labor cost or the likelihood of introducing bugs with complicated logic.

6.2.2 Document Store (MongoDB)

The leading open source document based storage engine is MongoDB. Document-oriented databases are essentially key-value stores that also provide additional 'relational' features based on object metadata. Ideally, this allows for high scalability while still providing a traditional SQL interface and query optimization via indexing (at least for MongoDB). However, several independent benchmarks show that both read latency and throughput for MongoDB are significantly worse than other NoSQL competitors, specifically due to architectural decisions that do not scale well, such as the single-instance MongoDB Router [12]. These benchmarks caused us to immediately discount MongoDB as too cost-inefficient for our peak provisioning.

6.2.3 Column Store (Cassandra)

The most promising alternative to Couchbase was a column store, Cassandra. Column-oriented databases provide the ability to read or write only specific data columns efficiently and have very flexible indexing and partitioning capabilities.

For our purpose, we considered Cassandra since it is a leading open source column store, with many vocal users. In some benchmarks, Cassandra outperforms Couchbase in terms of throughput per node and latency [7]. On the other hand, other benchmarks have shown Couchbase to provide

better performance and additionally, the previous paper used an older version of couchbase which is known to have worse performance than the current 4.x versions [2]. In any case, both linear scaling properties and the per node performance of Cassandra and Couchbase are comparable. Further discussion of benchmarks can be found in the Benchmarking section.

For the purpose of productionalizing the new CRS, the major issue with Cassandra was cost in developer time. In order to achieve the reported performance for Cassandra with our use case, fairly extensive data modeling and system tuning would have been required.We executed our own load tests with YCSB and found that Cassandra ran into issues while inserting large records, which Couchbase did not have. This is certainly a problem that can be solved with intelligent schema design, but out of the box a simple cassandra 'table' would not support our needs for storing several terabytes of large rows.

Cassandra uses a hash of the 'partition key' (a set of columns from a row) to determine on which node data is located. This causes some challenges when you want to retrieve data based on a date range, since the data is distributed across the entire cluster and you cannot take advantage of Cassandra's ordered storage of data on disk. DataStax does offer several data models that enable time range queries, but they have limitations of row size that would quickly be overwhelmed by 100KB crash reports [11]. While it is certainly possible to implement time range queries on large data sets in Cassandra, it requires a non-trivial amount of effort that quickly racks up cost in terms of developer time.

In addition, there are many configuration options for Cassandra that in some cases drastically change performance characteristics. This is a useful feature, since it allows for so much control; however several of these configurations become scaling bottlenecks with no manifestation of problems until the bottleneck is reached [10, 5].

Cassandra is certainly a highly scalable NoSQL database that provides much flexibility, both in terms of system tuning and indexing options. For many use cases it is a highly desirable storage solution. Nevertheless, we decided to use Couchbase since there was no obvious benefit from Cassandra for our use case. Setting up and maintaining Couchbase is much simpler due to the minimal configuration requirements, both have comparable throughputs and latencies in the CRS scenario and the developer effort required to integrated Cassandra added undue complexity.

6.3 Alternative Indexing Strategies

Before settling on the MySQL-based indexing strategy listed above, we considered two other options: using Couchbase 4.x's built-in Global Secondary Indexes, and using Elasticsearch.

6.3.1 Couchbase Indexing

The primary draw for using Couchbase GSIs was to allow the reporting API to issue SQL queries to Couchbase directly and remove the MySQL dependency completely. GSIs provide partitioned indexes that allow queries to scan a subset of document keys based on the provided partition columns. We did not even consider using a primary index since that requires a full bucket scan which, after even minimal testing, was clearly not viable with our 500 million document dataset. The GSI model we considered involved building one GSI per sku, since every API query requires a sku filter. Ultimately, Couchbase indexing was impractical because enabling it made the cluster's responsiveness highly intermittent, the read throughput was significantly lower and it required manual management of index creation and replication.

When only one index was created for one sku with our production data, we saw repeated failures of couchbase processes. On all of the servers the indexing and key-value processes repeatedly crashed and restarted due to timeouts or out of memory errors. While Couchbase does allow for setting memory limits on the key-value and indexer processes, it does not for the 'projector' process that forwards data between the key-value data stream and the indexer. We saw that this process was using an inordinate amount of memory and causing the system to fail. The only recommended solution we could find was to allocate more hardware and memory. Since we have several hundred skus active in our system and this issue presented itself while indexing merely one, it was clearly not cost-effective to allocate enough hardware to sustain the Couchbase indexer.

Furthermore, we saw drastically slower read performance when using Couchbase's built-in N1QL query service. Although indexing did not impact write performance (outside of the aforementioned cluster instability), it performed markedly worse than the key-value api in read throughput. Using 5KB documents, a single server could sustain 4500 key-value read operations per second, while it could only support 2200 GSI-backed read operations per second via N1QL. While 'read throughput' is not a common term for read evaluation, it is notable in that it sets the lower limit of the query latency. If only 2200 operations per second are supported, then a query returning 10000 documents will take at least 4.5 seconds to complete. Using N1QL would double our hardware requirements to support the same worst-case latency for large queries.

Finally, using Couchbase GSIs was not even remotely practical from an administrative point of view. Couchbase does not have any support for automatic index creation or replication. We would need to monitor any new data coming in for new skus and initiate index creation at that point in time. Additionally, we would need to build replica indices in the case of node failover; N1QL does handle reading from the correct index in the case of failover, but Couchbase doesn't support the auto-generation of replicas. Index creation also requires a full bucket scan, which ends up taking hours to days of time and requires a large amount of disk space (about 5 * final index size) due to fragmentation that only gets cleaned up after creation is complete. The final implications of this were that we would have to build quite a large monitoring and automation system to ensure that all required indexes were always present and replicated; even then, the latency required to build new indexes means that data for new games would be un-queryable for some time.

6.3.2 Elasticsearch

Elasticsearch is open source software built on top of Lucene and provides a scalable full-text search engine [6]. Couchbase provides an Elasticsearch plugin which utilized Couchbase's cluster replication feature to load data into an Elasticsearch cluster. The primary issue with this model is that indexing occurs completely asynchronously; we cannot guar-

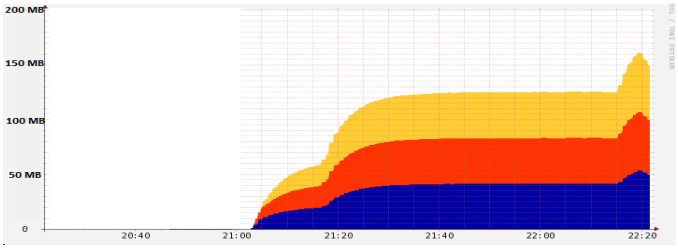

Figure 5: Incoming Network I/O Utilization of Kafka Servers (MB/s)

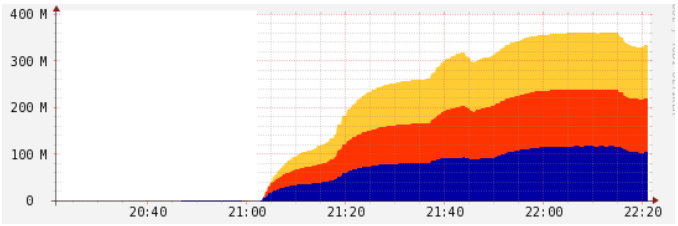

Figure 6: Outgoing Network I/O Utilization of Kafka Servers (MB/s)

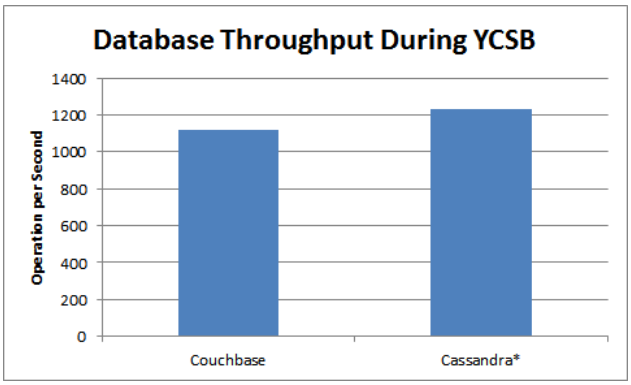

Figure 7: Couchbase vs. Cassandra Throughput with Worst-Case CRS Workload

antee what, if any, documents have been properly replicated to the Elasticsearch cluster. Consistency is one of the Legacy CRS concerns that we wished to address and so this eventual consistency model was too loose for the our current use case. We are investigating elasticsearch as a tool for enabling full text search on crash reports but that will be detailed in the 'Future Work' section of this paper.

7. BENCHMARKING

We executed synthetic benchmarks using sample data from our production environment and in some cases synthetic data that mirror production scenarios. We were most concerned with realistic benchmarks for the event capture process since insufficient scale for those would imply data loss; inaccurate provisioning of downstream components would only affect user experience. These benchmarks give us average case latencies and maximum throughputs that can be supported by a given amount of EC2 hardware. All tests were executed in a Virtual Private Cloud in AWS EC2, using m3.xlarge instances.

7.1 Capture Process

7.1.1 Methodology

For tests with production data we used several EC2 m3.xlarge instances to download past requests from S3 and replay them. We generously overprovisioned load generators to ensure they did not produce false bottlenecks. As mentioned previously, there are several data taxonomies which the capture service accepts; the following benchmark results describe isolated tests where only one data stream was replayed at a time.

7.1.2 Throughput

As you can see from the results in Table 1, for crash report payloads, the entire 100 MB/s network bandwidth was utilized, thus indicating that these tests measure accurate ceilings on throughput. With this model, we predicted that the capture service was capable of publishing approximately 1,800 crash reports or 15,000 session events per second to Kafka per server. We used this to estimate our production capture server requirements in the provisioning section.

7.1.3 Kafka

We used the same testing methodology to prove that Kafka is strictly network I/O bound in this environment; in addition we ran a consumer process that read data from Kafka to produce both read and write load. We used m3.xlarge instances for Kafka, with a three node setup and triplicate replication enabled. Figures 5 and 6 represent the three Kafka servers' inbound and outbound network usage over the course of the benchmarking test, respectively. These graphs represent the usage as we slowly increased the test traffic until we reached a peak steady state of approximately 90MB/s (or 30MB/s per Kafka broker) as seen in Figure 5. At this point, each broker was utilizing approximately 120 MB/s of total network bandwidth (which was derived by adding the values in Figures 5 and 6). When we attempted to increase the throughput (at time 22:15 in the aforementioned graphs), replication failures in the Kafka cluster began to occur, presumably due to the network I/O ceiling.

Since Kafka needs to use I/O to receive, replicate and send data, the 4:1 ratio of I/O usage to data volume makes natural sense. One part is used for receiving the data stream, two parts are used to generate the two other replicas and one part is used for client consumption.

7.2 Couchbase and Cassandra

Although there are many YCSB results for NoSQL databases, we executed our own since none of those tests use documents larger than 1KB. We used documents of size 100KB and ran the YCSB using m3.xlarge instances for both the clients and servers; these tests represent the performance using only a single server and without replication. Additionally, most other tests do not cover situations where data does not completely fit in memory; the above tests represent storing 50% of data in memory. We attempted to reflect a scenario where nearly no reads occured in memory; however when loading data into Cassandra we experienced many timeout errors that made this impossible. During the actual tests, this also caused timeouts on insert operations for Cassandra.

Table 1: Capture Service Benchmark

API	Throughput (Requests/s)	Average Latency (ms)	Network Usage (MB/s)	CPU Usage (%)
session events	15,000	16.0	34.9	98
crash reports	1800	33.3	108.2	98

Table 2: YCSB Latencies with 100KB Records

Database	Op	Min (ms)	Max(ms)	Avg(ms)
Couchbase	Read	0.80	138	7.45
Cassandra*	Read	0.80	562	18.7
Couchbase	Insert	2.6	365	11.3
Cassandra*	Insert	1.47	2000	6.80

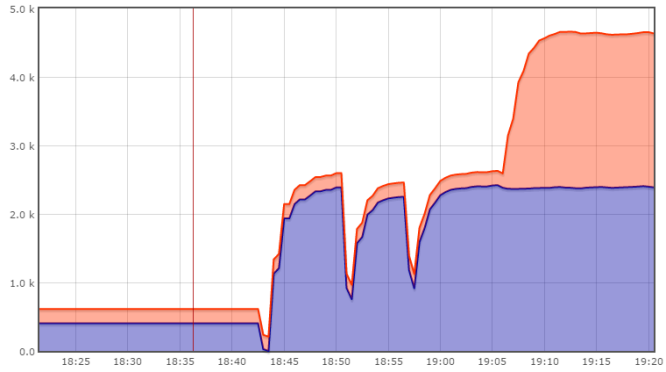

Figure 8: Session and Crash Events Persisted Per Second by Instance

For these tests, our goal was to determine if either database significantly outperformed the other. Table 2 shows that, in terms of latency Couchbase outperforms Cassandra on average for read operations and Cassandra outperforms Couchbase. The startlingly high 2,000 ms maximum latency for Cassandra, in conjunction with the aforementioned errors, indicated to us that Couchbase would perform more consistently out of the box. Figure 7 also shows that Cassandra executes approximately 100 more operations per second than Couchbase. Neither the differences in average latency or throughput signaled a clear winner to us; however, the fact that Couchbase provided more reliable performance became one of the deciding factors in this evaluation.

7.3 Consumer

7.3.1 *Methodology*

We ran two integration load tests with the consumer; one with normal CRS traffic and one with a data stream that consisted only of crash reports. The former represents the day-to-day throughput capability, while the second is a 'worst-case scenario' that could occur during a severe crash spike. For each load test, we preloaded a Kafka topic with several GB of production data to ensure that data availability in Kafka would not be a bottleneck.

For the normal data stream, we started a single m3.xlarge consumer instance, waited 10 minutes and then started a second instance; the results of that test can be seen starting at minute 19:00 in Figure 8. The earlier peaks in the graph represent starts and restarts of the consumer to confirm functionality prior to the load test and each color represents the throughput of one instance. Since we see equal throughput rates for both instances, both before and after starting a second, we can infer that the bottleneck to throughput during this test was indeed the consumer and not Kafka or Couchbase.

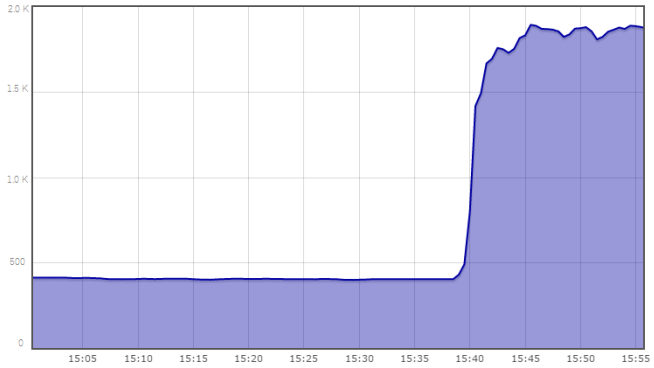

Figure 9: Crash Reports Persisted Per Second

For the crash-only test we repeated the same procedure; Figure 9 only displays the throughput of a single consumer instance.

7.3.2 *Results*

Figure 8 shows that the consumer can persist 2,300 events per second from the normal mix of event types. If all events are crash reports, each consumer can processt about 1,700 crash reports per second, which also translates to approximately 8.5 MB of data per second. This scales linearly with number of consumer instances until read parallelism from Kafka or write parallelism to Couchbase becomes a bottleneck.

7.4 Reporter

7.4.1 *Methodology*

To benchmark the reporter we ran multiple sets of tests for latency on both the Legacy and Couchbase-backed implementations. We varied both the size of the quries (measured in number of reports returned) and the number of concurrent queries. Network I/O is an uncontrolable factor for both systems, since clients operate in different network environments than either the Legacy data center or AWS. In order to account for these network disparities, we measured time to first byte (TTFB) responses. Each test was run 5 separate times and all latency results were averaged together. All queries were made using the same *sku*, which averaged reports of size 100 KB.

The Legacy Reporter was run on a single CentOS 6 machine, which has 16 cores and 42 GB of RAM. It was con-

CRS	1 Report	200 Reports	2,000 Reports	20,000 Reports
Legacy	0.10	0.20	1.42	2.76
New	0.17	0.51	2.34	25.0

Table 3: Latency for Reporter Queries of Varying Size - Simple Fields Only (sec)

CRS	1 Report	200 Reports	2,000 Reports	20,000 Reports
Legacy	0.14	1.73	4.57	23.9
New	0.17	0.51	2.26	24.7

Table 4: Latency for Reporter Queries of Varying Size - All Fields (sec)

nected to a MySQL 5.7 database, running on a CentOS6 machine with 24 cores and 94 GB of RAM.

The New Reporter was run on a single m3.xlarge instance. It was connected to two m3.xlarge Couchbase servers.

7.4.2 Results

Tables 3, 4 and 5 compare the latencies of queries, varying in size between 1 report returned and 20,000 reports returned (100KB to 2GB of data). Table 3 refers to queries which exclude fetching blob and media data from the reports; in this case the amount of data returned is only a few hundred bytes per report. Table 4 refers to normal queries, which return the full report data. Table 5 refers to when 10 concurrent queries were run of the specified size.

The Legacy MySQL implementation was by far superior if the API was required to only return the simple field set, since Couchbase requires fetching the entire document, even if only some fields are returned. In addition, the MySQL implementation was slightly better performing in the case where the queries were very large; however this was most likely due to the vastly superior hardware used in the Legacy Reporter.

The new implementation outstripped the legacy implementation in all situations where multiple queries were issued simultaneously and where a smaller number of reports were queried.This does suggest that client behavior needs to be modified, such that they issue many small queries in parallel. A *limit* and *offset* system such as that used in SQL has been incorporated into the reporting API to encourage such behavior.

It should be noted, that in the case where many MB of data are queried (such as with 2000 or more reports), I/O will be the dominant factor in latency. The average network bandwidth allocated per connection for the network between client and reporter is no more than 1 MB/s; as such the 25 or 29 seconds to query 2000 reports is dwarfed by the 100 seconds taken to transfer the data. However, this network bandwidth is limited per connection and not per server; issuing multiple small queries will help to alleviate this issue for clients and further reinforces the advantages of the new approach.

CRS	1 Report	200 Reports	2,000 Reports
Legacy	0.85	4.32	29.78
New	0.99	1.96	23.2

Table 5: Latency for Reporter Queries of Varying Size - 10 Concurent Queries (sec)

8. PROVISIONING AND COST

Based on the above performance metrics, the CRS use case needs 7 m3.xlarge instances and 1 m1.xlarge instance to process normal traffic volume: 1 capture server, 1 Kafka broker, 1 consumer, 1 reporter api server and 3 couchbase servers. In fact, for most of those services, the hardware will not be fully utilized.Additionally, one mysql master-slave pair is retained from the Legacy CRS; however it is likely overprovisioned for its new use and will be scaled down in the future to a small piece of hardware.

In reality, this small number of machines is not sufficient to provide a highly available and reliable service; neither does it include provisioning for data spikes nor does it account for replication of data in Kafka.

8.1 Reducing Overhead

The daily CRS workload is not massive; it is only during spike events when the traffic volume becomes a significant issue. We need to provision for that event since that is when reliability and scale is key, and we cannot entirely predict it ahead of time. However,the capture service, Kafka and Couchbase are are all multi-tenant in our ecosystem; this means that CRS is not the only application to use those resources. In general, the capture service and kafka are already provisioned with headroom to ensure no data loss for other taxonomies already. This headroom is already enough to handle 5-6 times the volume of normal CRS traffic. Since crash spikes are so short lived, and other data streams have much more predictable patterns, there is little risk that total usage will exceed this headroom. Either the crash spike will be processed before any other issues can occur, or there will be enough warning to provision more capture servers to compensate for the issue (rising traffic, cloud availability zone outage, etc.)

In this way, it is justified to claim that this design does not need to pay the costs of high availability; they are essentially one-time costs paid by the organization that become amortized across services.

9. PRODUCTION PERFORMANCE

The New CRS has been deployed on the aforementioned hardware and has been running as part of our production environment for several months. Not only has this project been a theoretical proof of concept, but it has withstood the test of several prominent game launches and open betas with minimal issue. The average time elapsed between crash submission and crash report availability is 600 ms and the 95^{th} percentile for this latency is 950 ms. Specifically, Couchbase performs significantly better than anticipated, with a 3 ms average insert latency and supports 2,000 operations per second per server; the majority of the latency comes from publishing data to and consuming data from Kafka.

As far as dataloss, reports are dropped occasionally due to various temporary glitches. Figure 10 presents a view of all such events over the course of one month in terms of reports dropped per second from the real-time system. Very rarely do such glitches cause data loss of more than 10 events per

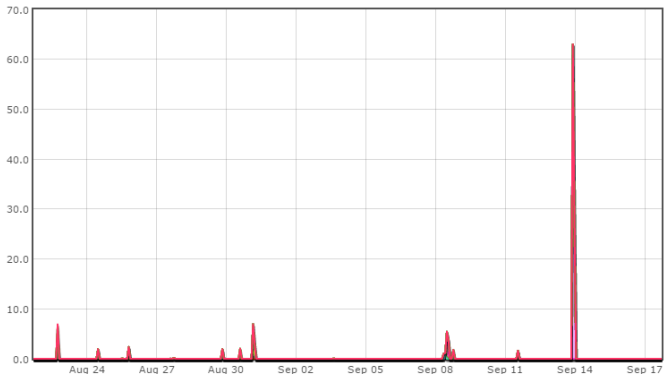

Figure 10: Events Dropped Per Second over 1 Month

second and all such issues are resolved within approximately 10 seconds. The largest spike of 62 events per second was caused by a hardware failure and bug which caused retries to be incorrectly attempted; this issue was caused merely by implementation, not a design flaw.

10. CONCLUSIONS

Overall, the New CRS improves or maintains data reliability, data availaibility and query responsiveness, while reducing cpu and memory usage.

10.1 Data Integrity

With the Legacy CRS, there were numerous incidents where we lost millions of of session events and tens of thousands of crash reports. While the Legacy CRS lacked the requisite monitoring to get exact figures, it would routinely lose millions of session events and hundreds of thousands of crash reports. With the new CRS, there is still occasional data loss; however it is mitigated to very small periods of time due to temporary errors that exceed the duration of capture service retries. Furthermore, none of that data is permanently lost, it is merely lost from the near-real time reporting of the CRS and is still reprocessable. This is very clearly an improvement over the Legacy CRS.

10.2 Latency

Next we can compare the query latency and end-to-end persistence latency of the Legacy and New CRS. The end-to-end latency is the total time elapsed between report capture and data being visible to the reporter api. For the Legacy CRS, it is on the order of 1 to 2 seconds in the normal case. The new CRS achieves a steady 550 ms average and 900 ms 95th percentile end-to-end latency. During load spikes, the end-to-end latency will rise if the provisioned consumer processes cannot meet the incoming crash rate. This is the tradeoff that we have made to ensure that data is eventually persisted[2].

For query latency, the load testing section above clearly demonstrates the general superiority of New CRS. The most common scenario in production is around 10 concurrent user queries, which average 20 to 100 reports in size; the New CRS responds to those queries with a 54% decrease in latency. In the case of very large queries, the New CRS did perform slightly worse with a 3% increase in query latency. However, the New CRS is arbitrarily horiztonally scalable and thus such poorly performing queries can be improved simply by splitting them into smaller concurrent queries.

10.3 Resource Usage

As mentioned above, the new CRS needs only provision 6 m3.xlarge instances and 1 m1.xlarge instance, thanks to amortizing the cost of high availability across other services. The Legacy CRS used 8 collection servers, 3 MySQL master-slave pairs and a single reporter server. All of these had varying specifications, which were invariably larger than the m1 and m3 instances provisioned in AWS. In total, the Legacy system used 208 CPUs and 710 GB of RAM: 48 cores and 100GB for collection; 144 cores and 564 GB for MySQL; and 16 cores and 46GB for reporting. The New CRS uses 80 cores and 308 GB of RAM: 8 cores and 30 GB for the capture process; 40 cores and 154 GB for data persistence; and 4 cores and 15 GB for reporting. This represents a 63% decrease in CPU usage and a 59% decrease in memory usage.

10.3.1 Other Costs

While it is less quantitative, the time spent to maintain these systems should also be taken into account. The New CRS is built on cloud infrastructure that survives without manual interaction even in the event of node failure. This also allows for much quicker turn around on server maintenance, scaling and configuration. Furthermore, we have found couchbase to be a very stable system that requires little configuration or headache to maintain, as long as its N1QL indexing is not enabled. It is difficult to put a price on these advantages, but they have certainly allowed us more rapid and reliable testing and provisioning of hardware than if we were still using a proprietary data center.

11. FUTURE WORK

At the moment, the reporting API is very bare and can only optimize queries based on time ranges. We would like to continue to investigate other methods for asynchronous indexing, either by building an in-house system using rdbms for index storage (as we have primitively implemented here) or by taking advantage of open source software such as Lucene and Elasticsearch. For either of these approaches our primary concerns are focused on enabling full-text search and creating an index service that scales both with the amount of incoming data and the number of index scan requests. Furthermore, there are more demands from game developers to increase payload size by including even more detailed stack traces, higher resolution images and/or short video clips. We seek to support these changes by evaluating the impact of storing even larger media files on Couchbase and how best to efficiently process such bulky reports, at protocol and infrastructure levels.

[2]For the current provisioning, a 5 minute crash spike reaching 3300 reports per second (over a 10X rate spike) will incur a worst case latency no more than 5 minutes.

12. REFERENCES

[1] F. 7. Benchmarking network performance of m1 and m3 instances, 2014.

[2] Altoros. The nosql technical comparison report, 2014.

[3] I. Amazon.com. Amazon web services, 2016.

[4] Apache. Apache kafka, 2016.

[5] S. Bisbee. Scale it to billions - what they don't tell you in the cassandra readme, 2015.

[6] E. Co. Elasticsearch, 2016.

[7] E. P. Corporation. Benchmarking top nosql databases, 2015.

[8] Couchbase. Couchbase, 2016.

[9] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[10] Datastax. Cassandra configuration, 2016.

[11] Datastax. Getting started with time series data modeling, 2016.

[12] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser. Performance evaluation of nosql databases: A case study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, pages 5–10, New York, NY, USA, 2015. ACM.

[13] I. B. Times. Battlefield 4: Bugs and issues lead to backlash, 2013.

[14] R. Wu. Diagnose crashing faults on production software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 771–774, New York, NY, USA, 2014. ACM.