Market Reconstruction 2.0: A Financial Services Application of Google Cloud Bigtable and Google Cloud Dataflow

Neil Palmer, Sal Sferrazza, Sebastian Just, Adam Najman - FIS
{neil.palmer; salvatore.sferrazza; sebastian.just; adam.najman}@fisglobal.com

Abstract

The FIS Market Reconstruction Platform (MRP) is a market surveillance and forensics utility designed for the daily ingestion, linkage and analytics of U.S. equity and equity options market events over multi-year periods. In May of 2015, FIS' Advanced Technology group (formerly SunGard) released a white paper detailing its experiences using Google Cloud Bigtable in prototyping the MRP. In the months that followed, Google opened several other services within its Cloud Platform suite to general availability (GA), among them Google Cloud Dataflow. The solution architecture is revised to include both Cloud Dataflow and Google's BigQuery service in addition to Bigtable. With the combination again considered in the context of the MRP requirements, we conducted additional experiments measuring the speed and scalability of the revised architecture. We have found that while it is imperative that engineers adopt specific approaches that capitalize upon cloud and parallel computing paradigms, employing these services together does yield the potential for extraordinary computing performance.

1 Background

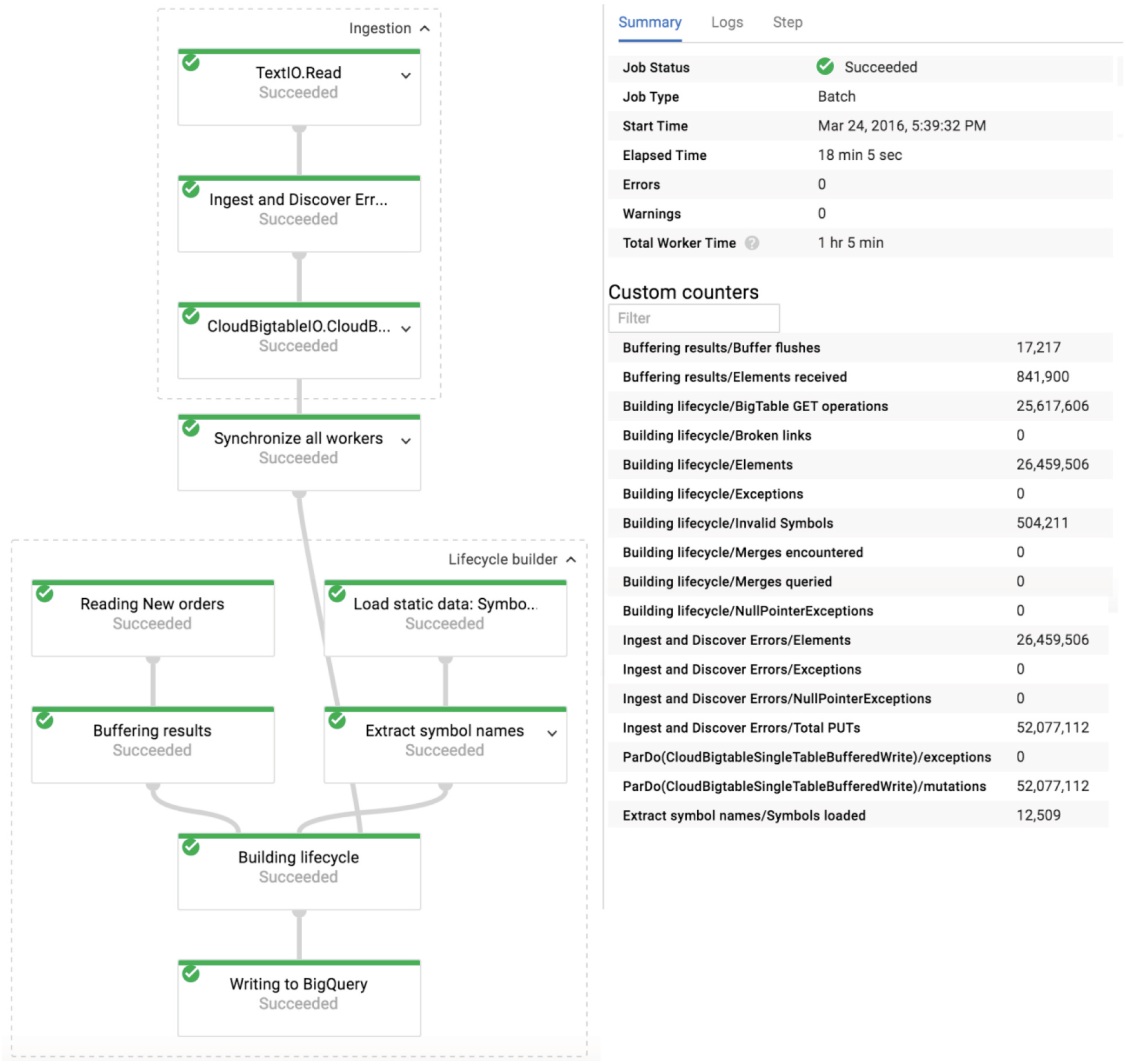
1.1 Introduction

As part of its bid to become the Plan Processor for the Consolidated Audit Trail (CAT) initiative,[1] FIS developed a prototype of the principal order linkage use case specified by CAT's requirements. In May of 2015, FIS detailed the results from the initial version of this prototype, which was based broadly on Hadoop and Google Cloud Bigtable.[2] Since then, Google Cloud Dataflow has gone GA.[3] Here, FIS outlines its experience integrating Dataflow and BigQuery into its original prototype assets, and contrasts the relative observations of the two solution architectures from both development practices and runtime performance perspectives.

1.2 Reconstructing the market

While the principal functional requirements[4] of the linkage algorithm have not changed from 2015, several major components of the MRP's overall solution architecture have been replaced. Namely, the job orchestration and execution capabilities that formerly employed custom Java code running within Hadoop[5] have been ported to pipelines developed using the Google Cloud Dataflow SDK, and executed within the Google Cloud Dataflow runtime environment. The benefits that Cloud Dataflow offers developers are manifold; among them are: reduced infrastructure administration, automatic scaling, and an SDK with native support for both batch and streaming modes. Additionally, linkage results produced by the pipeline are now published to a Google BigQuery dataset, allowing for the visualization and analysis of derived lifecycle data immediately after processing.

1 For more information on SEC rule 613, see http://www.catnmsplan.com
2 Palmer et al. (2015) "Scaling to Build the Consolidated Audit Trail: A Financial Services Application of Google Cloud Bigtable"
3 "Google Cloud Dataflow and Cloud Pub/Sub Reach General Availability" – http://www.programmableweb.com/news/google-cloud-dataflow-and-cloud-pubsub-reach-general-availability/2015/08/18
4 SEC Rule 613 (Consolidated Audit Trail) – https://www.sec.gov/divisions/marketreg/rule613-info.htm
5 http://hadoop.apache.org/



Job Status	✓ Succeeded
Job Type	Batch
Start Time	Mar 24, 2016, 5:39:32 PM
Elapsed Time	18 min 5 sec
Errors	0
Warnings	0
Total Worker Time	1 hr 5 min

Custom counters

Buffering results/Buffer flushes	17,217
Buffering results/Elements received	841,900
Building lifecycle/BigTable GET operations	25,617,606
Building lifecycle/Broken links	0
Building lifecycle/Elements	26,459,506
Building lifecycle/Exceptions	0
Building lifecycle/Invalid Symbols	504,211
Building lifecycle/Merges encountered	0
Building lifecycle/Merges queried	0
Building lifecycle/NullPointerExceptions	0
Ingest and Discover Errors/Elements	26,459,506
Ingest and Discover Errors/Exceptions	0
Ingest and Discover Errors/NullPointerExceptions	0
Ingest and Discover Errors/Total PUTs	52,077,112
ParDo(CloudBigtableSingleTableBufferedWrite)/exceptions	0
ParDo(CloudBigtableSingleTableBufferedWrite)/mutations	52,077,112
Extract symbol names/Symbols loaded	12,509

Figure 1: Dataflow Pipeline: Execution Graph and Metrics

Figure 1 depicts a visualization of the end-to-end Google Cloud Dataflow pipeline for the ingestion and linkage components, with the major architectural pillars detailed in the subsequent sections.

1.2.1 Message ingestion

The ingestion component of the Dataflow processing pipeline takes input from synthetic Financial Information eXchange (FIX) protocol messages that reside in flat files within Google Cloud Storage, and performs syntax and structural validations upon them. After validation, specific FIX tag values are extracted from the submission files and loaded into Cloud Bigtable.[6] As a durable, distributed and shared memory repository, Bigtable is used to maintain a consistent, prevailing state of the trading session's order lifecycle graphs.

One of the MRP's operational constraints is that order event data can arrive from many disparate, independent market participants – with no chronological guarantee of arrival time, and therefore, no guarantee of processing order. When any individual event is processed, a complete end-to-end order lifecycle linkage will not be immediately possible if its subsequent or prior lifecycle events have not arrived.

Dataflow provides exactly once processing within the pipeline, yet side effects, like a write to Bigtable may be repeated unless managed properly. In order to parallelize event graph generation and engineer resiliency to individual worker failure,[7] Bigtable is continually updated with the results of all worker processing. That is, for any given event being evaluated within a Dataflow pipeline, all worker output is persisted to Bigtable.[8] If the same event happens to be processed multiple times

6 For example: 37 (OrderID), 54 (Side), 55 (Symbol) and 30 (LastMkt)
7 This is discussed more in section 2.2.3

because of worker failure (i.e., multiple `processElement()` invocations with identical `ProcessContext`), then it will ultimately derive to an identical record within Bigtable, as opposed to creating a duplicate event with identical properties.

This characteristic of random event arrival and inevitable worker transience is largely what drives the need for a durable, distributed persistence tier. By mere virtue of the total number of workers that must be deployed to accommodate the data volumes in accordance with service level agreement (SLA) constraints, arbitrary worker failure is almost a statistical certainty.[9] The platform must be resilient to these failures and ensure that the persistent tier is left in a consistent state at all times. It is in this area in particular where Bigtable's strength becomes clear. By capitalizing on Bigtable's native idempotency characteristics, the processing algorithms are free to be massively parallelized in their execution. No further synchronization points are necessary once a particular mutation has been confirmed by Bigtable. Using Dataflow and Bigtable together, system capabilities may be expressed as pure functions, left to execute under Dataflow's purview with all cumulative progress kept consistent by Bigtable and with in-flight failures retried automatically.

Prior observations of this stage indicated that the ingestion process places high demand upon I/O capacity. As in most platforms that scale hardware resources horizontally, meeting the demand for a particular processing SLA is achieved by allocating additional segments of the input data universe across more worker nodes.[10] The ingestion subsystem of this latest testing effort, originally developed using the HBase API and deployed with Hadoop, was ported to use the abstractions and facilities of the Cloud Dataflow SDK (now Apache Beam) and the Google Cloud Dataflow runtime environment.[11]

In contrast to the combination of Hadoop and HBase API originally used, applying the Dataflow SDK and cloud runtime environment to the workload yields several clear advantages for development teams. These include:

- Decreased infrastructure management footprint
- Automatic scaling of worker nodes based upon velocity of input data
- Additional diagnostics and insight into worker operations via the Google Cloud Console

Bigtable's fundamental column-oriented model means that mapping normalized relational schemas one-to-one is neither practical nor recommended. Instead, Bigtable employs the concept of "column families,"[12] which is fundamental to many NoSQL schema designs.[13] This means that all entity properties reside within a single table, and any relational joins between entities must be performed by the application programmer. Stylistically, entity properties tend to be self-contained in a single "Big" table. Table 1 illustrates the schema employed for the event linkage process. In principle, FIX messages are less arduous to persist to NoSQL-styled databases vs. relational databases, since the FIX data dictionaries (i.e. schemas) themselves employ quite a bit of nesting and relationships at the individual entity level.

Column Family	Columns
Data	• Quantity • Source • Destination • Symbol • ChildSymbol • SymbolValid • SymbolMatch • Side • ChildSide • SideMatch
Link	• Parent RowKey • Child RowKey • Merge RowKey • Grandparent ID
Error	• Error Status

Table 1: Simplified Bigtable Schema

For the chosen Bigtable schema, entity data and the necessary entity relationships are kept within separate column families. The `Data` column family contains the persisted tag values from each validated FIX message. The `Link` column family contains the information necessary to derive the event relationships, represented in a way that reflects the industry's own mechanisms for designating counterparty relationships.[14]

The ingestion process enforces valid FIX messages and subsequently loads the relevant FIX tag values into their respective destinations within the Data column family.

Validations against externally maintained datasets are also performed on the input FIX messages during this stage, such as validating that events use identical reference symbology for instruments and participant identifiers.

These validations generally fall into one of three separate categories:

- Structural correctness of submission syntax (in this case, FIX 4.2)
- Parent/Child event matching based on a specific property of the original data (i.e. parent and child order identifiers)[15]
- Contextual validation, such as validating that a participant is recognized and that no messages are using deprecated symbology

First, the formal correctness of data is confirmed during ingestion, such as validating that the value submitted for a particular numeric data field is devoid of alpha characters, and storing the result in the *`Valid` column.

8 Specifically, the output of a `ParDo`'s implementation of `DoFn.processElement()` as defined in the Dataflow SDK
9 Dean (2008)
10 Palmer, et. al (2015) supra
11 The Google Cloud Dataflow SDK was recently accepted as incubating by the Apache Foundation, and renamed to Apache Beam
12 "Designing Your Schema" – https://cloud.google.com/bigtable/docs/schema-design
13 "Column (Family) Databases" – https://ayende.com/blog/4500/that-no-sql-thing-column-family-databases
14 CAT NMS Plan - "SRO NMS Plan Industry Call" at http://www.catnmsplan.com/web/groups/catnms/documents/catnms/p197375.pdf
15 Öhman (2013)

Next, the event's own data point is matched for equality with the child's data point. A success or failure is marked in the `*Match` column, allowing for the flagging of orphan events.

Finally, the data point might be checked against a set of rules or an external system to confirm its correctness. For example, the system must already know participant identifiers and should be able to identify and flag the use of deprecated symbology. The results of these contextual checks are also stored in the `*Valid` column.

These types of issues can therefore be iterated simply via queries to Bigtable on those Boolean columns. Additional details pertaining to the processing errors for that event are stored in the Error column family to simplify tracking and diagnostics.

1.2.2 Lifecycle construction

The output of the lifecycle construction stage is a graph of related market events that represent complete order lifecycles.[16] During linkage, the algorithm must address the limitation of each individual event possessing only a reference to its most direct parent, or a null reference indicating that the event represents the root of a particular order lifecycle.

Part of the lifecycle algorithm is responsible for maintaining the "root" node, or genesis event, which we refer to as the "grandparent," that ultimately represents the oldest ancestor of a complete order lifecycle.

This limitation emerges as an artifact of current U.S. equity market structure: by the time an order has been filled in its entirety, the overall accumulation of shares in order fulfillment will have originated from an indeterminate set of individual market participants. Additionally, parent or child orders may be cancelled (and/or replaced) in whole or in part before fulfillment. Furthermore, each participant is only knowledgeable of, and hence can only report on, its own direct interactions with individual counterparties per market event. By the time an order graph terminates, the component shares may have been aggregated and disaggregated with other orders several times.[17] To reconstruct the complete order lifecycle, a linkage strategy commonly referred to as the "daisy chain" model is used.[18]

While each non-root event is tagged with its parent `Link` column family, the same pattern is applied for the columns named `CHILD_*` in the `Data` family. This allows for simple data scans and parallel processing of event context. As each market event confirms that its children employ identical reference symbology, for instance, it can be confirmed that the whole lifecycle is predicated upon that same foundation. Since the information is grouped within the dedicated column family, the total amount of data transferred between Bigtable and the worker nodes performing these operations is minimized.

This seemingly minor optimization becomes significant considering the sheer number of worker routine iterations. Application programmers commonly demand more information from a repository than is strictly necessary for a particular use-case round-trip,[19] but when computing at the petabyte scale, efficiency is crucial. To put this into perspective, during our 60-minute, 25 billion-record test, saving one byte per record is the equivalent of reducing the overall sustained bandwidth requirements by nearly 56 Mbps:[20]

Given a single byte per record:

$$25{,}000{,}000{,}000\ \textit{records} * 1\ \textit{Byte} = 25{,}000{,}000{,}000\ \textit{Bytes} = 25e^9\ \textit{Bytes}$$

Processed per second over the course of an hour:

$$\frac{25e^9\ \textit{Bytes}}{60\ \textit{minutes}} = \frac{25e^9\ \textit{Bytes}}{3600\ \textit{seconds}} = 6{,}944{,}444\ \textit{B/s}$$

Converted to megabytes:

$$6{,}944{,}444\ \textit{b/s} = 6.944\ \textit{MB/s}$$

Converted to megabits per second:

$$6.944\ \textit{MB/s} = 55.5\ \textit{Mb/s}$$

16 "Understanding Order Execution" – http://www.investopedia.com/articles/01/022801.asp provides an introduction to order execution logistics from a retail trader's perspective
17 Öhman, (2013) *supra*
18 See SIFMA Whitepaper - "Industry Recommendations for the Creation of a Consolidated Audit Trail", see also CAT NMS Plan - "SRO NMS Plan Industry Call" – http://www.catnmsplan.com/web/groups/catnms/documents/catnms/p197375.pdf pp17
19 "10 More Common Mistakes Java Developers Make When Writing SQL", #2 – https://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql/
20 Google Search Results: " 25e9 bytes / 1 hour in Mbps" – https://www.google.com/?q=25e9%20bytes%20/%201%20hour%20in%20Mbps

1.2.3 Lifecycle publication

The original solution architecture did not specify any native mechanism for visualization of the derived lifecycles.[21] With this test iteration, given the introduction of Cloud Dataflow support for both Bigtable and BigQuery, we decided to include a stage near the conclusion of the pipeline that persists derived linkage results to a BigQuery dataset, in order to leverage its rich support for a broad array of third-party analytics.[22]

This was performed by flattening the Bigtable column families and only using the top level of column data (ignoring any underlying nested structures). For improved performance and an optimal footprint of at-rest data, BigQuery's native data types were used.[23]

The derived BigQuery schema is opportunistically denormalized. For example, the `CHILD_*` columns allow the analytics tool to outline potential issues with event submissions without resorting to joins or multi-table `SELECT` statements. The same principle applies to the `Error` column family, which provides additional details around why a particular event was flagged.

The following illustrates the output of `bq --pretty show` for the BigQuery schema used by Dataflow to facilitate post-processing analytics.

```
+-----------------+---------------------------------+------------+-------------+------------+
|  Last modified  |             Schema              | Total Rows | Total Bytes | Expiration |
+-----------------+---------------------------------+------------+-------------+------------+
| 24 Mar 11:52:33 | |- id: string                   | 26459506   | 6884653269  |            |
|                 | |- Parent: string               |            |             |            |
|                 | |- GrandParent: string          |            |             |            |
|                 | |- timestamp: string            |            |             |            |
|                 | |- EventId: string              |            |             |            |
|                 | |- ReporterId: string           |            |             |            |
|                 | |- Child: string                |            |             |            |
|                 | |- CustomerId: string           |            |             |            |
|                 | |- CustomerIdValid: string      |            |             |            |
|                 | |- side: string                 |            |             |            |
|                 | |- CHILD_side: string           |            |             |            |
|                 | |- symbol: string               |            |             |            |
|                 | |- CHILD_symbol: string         |            |             |            |
|                 | |- CHILD_ordType: string        |            |             |            |
|                 | |- eventType: string            |            |             |            |
|                 | |- brokenTag: string            |            |             |            |
|                 | |- ReporterIdValid: boolean     |            |             |            |
|                 | |- SideMatch: boolean           |            |             |            |
|                 | |- SymbolMatch: boolean         |            |             |            |
|                 | |- SymbolValid: boolean         |            |             |            |
```

21 Palmer, et. al (2015) *supra*
22 Simba ODBC – http://www.simba.com/drivers/bigquery-odbc-jdbc/
23 "Data types" – https://cloud.google.com/bigquery/preparing-data-for-bigquery#datatypes

2 Recapping previous efforts

2.1 Items outstanding from the original white paper

While the total volume of data submitted during 2015's testing campaign did not represent the full extent of a typical U.S. equities trading session, the solution architecture was observed to exhibit near-linear scalability.[24] In contrast, for the updated system, the target volume was escalated to 25 billion order events, with the goal of deriving the requisite linkages among them in a single hour.

This particular benchmark was selected because it represents nearly double the estimated "58 billion" daily events projected for "the world's largest repository of securities transaction data"[25] and also implies that a four-hour window would provide regulators with the ability to reconstruct market events without a protracted duration of interim processing.[26]

We first began the testing campaign targeting a volume of one percent of 25 billion messages end to end, with continual, incremental adjustments to key configuration parameters (such as worker machine type and count) performed between iterations. Insights emerged during each step. For example, by the 10 percent test, it became apparent that the mere act of logging debug messages presented us with a material degradation of algorithmic performance. Even if a particular logging implementation is implemented in linear time, every operation – no matter how trivial – does accumulate to yield a non-trivial footprint at this scale. Hence, eliminating non-essential code is an absolutely crucial step in maximizing the performance of a pipeline.

In order to provide high-fidelity inputs representing a cross-section of market activity, it was necessary for us to manufacture a data generator for fictitious events depicting the order lifecycles of diverse market participants. Preliminary versions of the tool offered simple procedures that generated an assortment of static lifecycles, varying only in their randomly generated prices, symbols and MPIDs.[27] As this mechanism evolved alongside the solution architecture, it has been consequently expanded to produce increasingly complex market event lifecycles. This generative process was constructed using a Markov Chain, allowing us to parameterize the likelihood of a given event at any stage of an order lifecycle.

BigQuery has continued to provide the primary method for querying and accessing the post-processed order lifecycles. A visualization front-end was prototyped to showcase user interface (UI) concepts that facilitated the analysis and visualization of the massive volumes under the MRP's purview. The following is a screenshot from the UI prototype that takes advantage of the BigQuery API within a single-page application (SPA), combining both the derived lifecycles and public market data to ultimately provide the means for reconstructing point-in-time market activity.

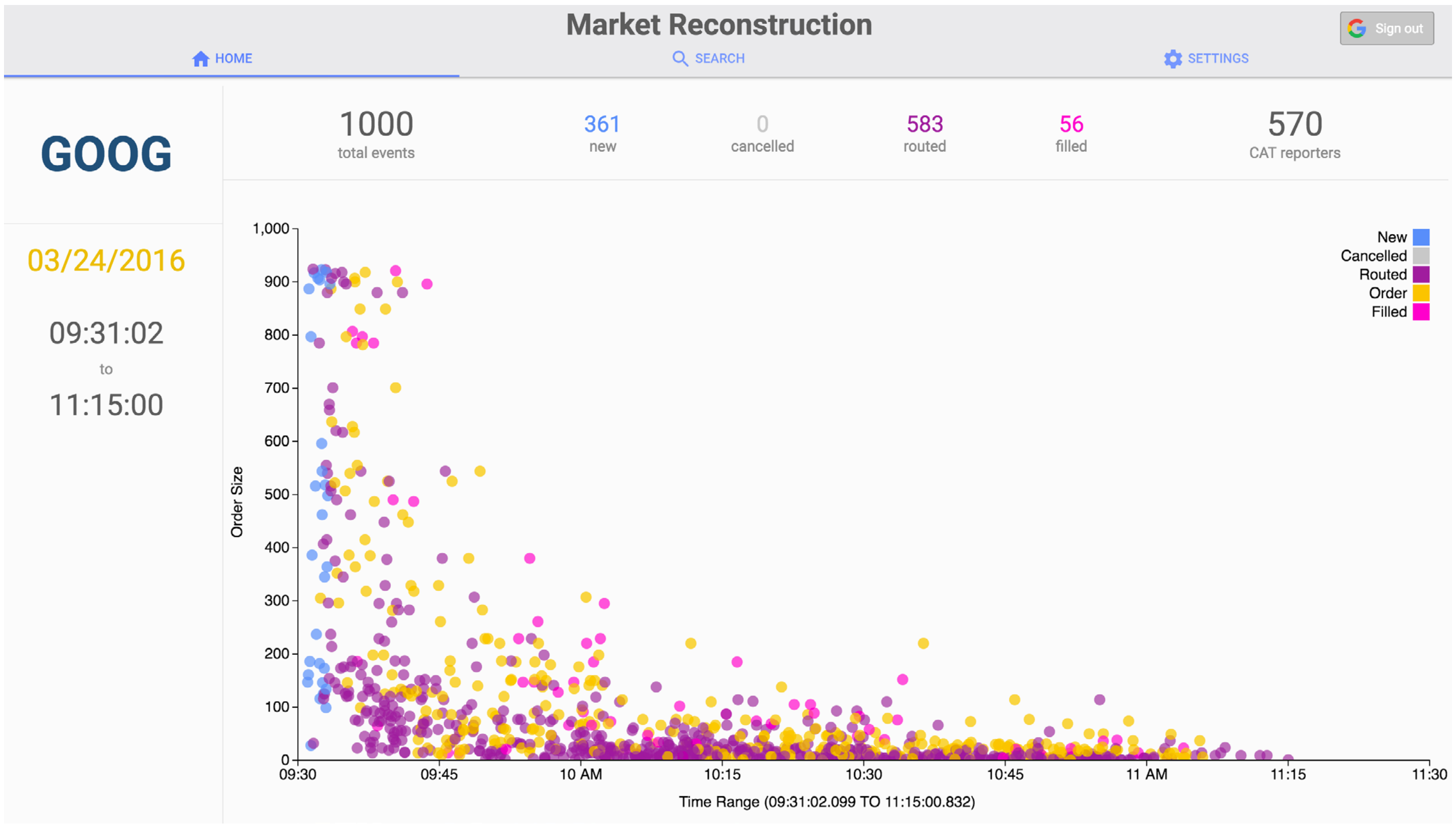


Figure 2: Market Reconstruction Scatterplot

24 Palmer, et. al (2015) *supra* pp 5
25 "Bidders for SEC's CAT System Narrowed to Six from 10" – http://www.wsj.com/articles/bidders-for-secs-consolidated-audit-trail-system-narrowed-to-six-from-10-1404247796
26 = SEC Rule 613 *supra*
27 A Market Participant Identifier (MPID) is a unique symbol identifying a specific trading unit registered with a national securities exchange or association. For example, the MPID for Morgan Stanley is "MSCO".

2.2 Addressing prior limitations

One of the motivations for revising MRP components was to address the limitations of the original solution. In this section, we describe how the addition of Google Cloud Dataflow and Google BigQuery to the solution architecture impacted the observed performance results and the overall experience in building and using the system.

The following sections correspond to the limitations detailed in our original white paper[28] and explain how they are addressed within the latest testing campaign.

2.2.1 Formalized testing of reads not completed

In 2015's testing campaign, measurement of Bigtable read performance was not explicitly broken out. During this exercise, however, measurement of read operations has now been added to the second stage of the pipeline. This stage builds the order lifecycles before publishing them into BigQuery.

2.2.2 Cannot possibly account for all possible order scenarios

We have increased the capabilities of our data generator to expose the lifecycle builder to a wider variety of lifecycle scenarios. We believe this yields a much more realistic simulation for reconstruction scenarios overall.

2.2.3 No evaluation of durability

When running parallel workloads at this scale, it is inevitable that some workers will fail midstream, for any number of reasons. The data graphs and worker population of the linkage process must be resilient amid both rapidly evolving state and worker ephemerality.

A particularly appealing feature of Dataflow is the ability to maintain a consistent worker population and preserve exactly-once within the pipeline processing guarantees when these inevitable worker failures are encountered. When a worker fails mid-pipeline, Dataflow will automatically restart the defunct worker process and re-execute the unit of work that was in-flight.

For its part, Bigtable is responsible for maintaining the durability and correctness of order lifecycles. Specifically, in the event that a portion of the Dataflow job fails and is subsequently retried, side effects, like writing to Bigtable are also retried, meaning a single record may be written to Bigtable multiple times. However, the final output is still correct, as Bigtable resolves these writes and returns only the latest version to downstream pipeline components. This is made possible by, and shows the importance of, having deterministic record UUIDs when using side effects in Dataflow.

2.2.4 Cluster sizing

Given the size of this load test and the methodical iteration practices employed, we are confident that we have arrived at an optimal overall configuration.[29] Observations indicate that the worker CPU-to-Bigtable node ratio scales linearly and optimizes the configuration for the respective compute and throughput-bound tasks.

The exercise[30] to determine optimal cluster sizing started with a fractional workload of one percent and assumed that all individual service resources would scale linearly. This assumption was predicated upon the belief that the custom business logic executed by Dataflow was free of locks, latches or synchronization points. The specific parameters used to instrument the various components from a sizing perspective were:

- Number of nodes in Bigtable cluster
- Dataflow worker machine types[31]
- Number of Dataflow workers
- Bigtable split ratio[32]

The ultimate goal when deciding on candidate configuration parameters is to yield maximum utilization of all component resources, but stopping short of queuing work at any tier. When scaling horizontally, additional resources are brought online when existing resources are at capacity.

All test iterations were closely monitored and fine-tuned. If resource metrics indicated that there was room for additional improvement, then there was always a prior baseline established for comparison. For example, Dataflow worker machine types were adjusted (while maintaining the same number of cores overall) in order to optimize the ultimate network footprint between Dataflow and Bigtable. By keeping granular performance statistics across all tiers of the stack, it was straightforward to evaluate specific tactical design approaches in the pursuit of an optimal configuration overall.

2.2.5 Failure to account for rebalancing and scaling of the Bigtable cluster

Bigtable will automatically attempt to learn data access patterns, and subsequently rebalance its region server definitions (also referred to as "splitting" the cluster). The HBase shell allows one to specify splits and hence influence the size of Bigtable's tablets.[33] For the purposes of this testing campaign, the split ratio upon which we ultimately decided was found to be sufficiently optimized.[34]

However, we expect that any ratios chosen would prove optimal only under specific, well-defined workload scenarios. Hence, organizations looking to conduct similar exercises must do so in the context of their own particular infrastructural constraints.

There exists a multitude of disparate configuration parameters that define each test's dimensions, such as worker node machine types, worker geography, and the disk size of cluster nodes. We found that managing these configurations diligently enables more precise measurements and higher-fidelity iteration comparisons, allowing one to truly pinpoint each tier's contribution to the overall resource footprint.

The resource measurements are not limited to simply the computing realm. One key economic benefit of a cloud computing model is the transparency of resource accounting, with the ability to access resource utilization programmatically. This allows teams to easily add the cost dimension to their infrastructure testing model, and use the resulting metrics to further inform architectural decisions, embedding operating expenditures among the metrics used to evaluate a particular architecture's overall suitability.

28 Palmer, et. al (2015) *supra* pp 8 table 4
29 See 3.2 Results
30 See 3.2.2 Resources consumed per target workload
31 See 4.4 VM sizing
32 See 4.7 Bigtable splits
33 Chang, et. al (2006) "Bigtable: A Distributed Storage System for Structured Data"
34 See 4.7 Bigtable cluster distribution and scaling

2.2.6 Bigtable's programmatic interface is limited to the HBase API

Using the HBase API enabled us to leverage existing tools (in this case, the HBase shell) to create and optimize our tables. One way we accomplished this was by passing in the region split parameters when the tables were initially created.

In late 2015, Google released a Cloud Bigtable I/O connector for the Dataflow SDK.[35] This capability gave the FIS engineering team the ability to integrate Bigtable and the Dataflow execution pipelines in a more standardized fashion. The previous design used the HBase API to communicate with Bigtable from within custom pipeline code. In contrast, representing Bigtable as a Dataflow I/O connector reduces some risk of abstraction leakage[36] and, given the standardized interfaces and API model employed by the Dataflow SDK, promotes reusability, modularity and standardized interfacing among the project's customizations.

2.2.7 No ad-hoc SQL for analytics

Recently, there has been a trend of product releases offering SQL-compliant APIs for interacting with NoSQL database platforms.[37] Given the existing installed base and popularity with end users, support for SQL as a query API is desirable.[38] Yet, until recently, the modern NoSQL landscape had offered limited SQL interfaces, instead preferring custom programmatic APIs or resorting to an implementation of "SQL-like" capabilities.

At the time of writing, Bigtable supports the standard HBase API for issuing queries, but no direct support for queries written in a SQL or quasi-SQL syntax.[39] However, our ultimate vision for market reconstruction and forensic activities indeed includes a SQL interface at the "last mile" in order to facilitate the independent efforts of business analysts and market regulators, and to capitalize upon its broad familiarity among industry practitioners.

Google's BigQuery managed database service, on the other hand, employs a dialect of SQL that analysts should find quite familiar. BigQuery also provides many analytical functions directly, beyond those observed in common vendor-specific SQL dialects. Even though still in Beta, Standard ANSI SQL is supported, too. In addition, BigQuery provides a mature REST interface secured by Google Authentication. This interface has been extremely helpful in the development of front-end analytic tooling without the need to deploy a full-blown application server tier between the browser and the event repository.[40]

35 "Dataflow Connector for Cloud Bigtable" – https://cloud.google.com/bigtable/docs/dataflow-hbase
36 Spolsky (2002)
37 Tools such as Apache Phoenix, Crate.IO and Apache Drill are some examples. The latter draws inspiration from Dremel, Google's own proprietary ancestor to BigQuery.
38 "Why SQL Is Important" – VoltDB – https://voltdb.com/why-sql-important
39 Cloud Bigtable and the HBase API – https://cloud.google.com/bigtable/docs/bigtable-and-hbase
40 Palmer, et al. (2016)

3 Scaling up

3.1 Approach, sizing and assumptions

To reach the target throughput goal of 25 billion events processed per hour, multiple runs of the one percent (250 million messages) workload were executed. This drove our estimation of the necessary resources required for the execution of a 100 percent workload.

Individual Google Cloud Platform (GCP) services may possess their own service-specific limitations. For example:

- Maximum compute allocation per Dataflow job: 10,000 cores
- Maximum bucket size from which `BigQueryIO` can import data: 10,000 objects

The observations illustrated in Table 2 represent averages derived from multiple test iterations for the particular resource configuration indicated.

3.2 Results

3.2.1 Executed runs

Target workload (%)	# Messages (billions)	# Bigtable nodes	# Dataflow worker cores (machine type)	Ingestion duration (minutes)	Lifecycle generation (minutes)	BigQuery import (minutes)
1	0.25	20	80 (n8)	32	35	
5	1.25	175	400 (n8)	20	36	
10	2.50	350	320 (n32)	35	40	
20	5.00	700	960 (n32)	25	n/a[41]	
40	10.00	1,400	1,920 (n32)	35	16	120
80	20.00	2,800	7,680 (n32)	34	15	180
100	25.00	3,500	9,600 (n32)	32	18	18[42]

Table 2: Test Campaign Metrics

41 The second part of the pipeline was never successfully executed to completion, therefore no time measurement exists
42 See 4.6 BigQuery import

3.2.2 Resources consumed per target workload

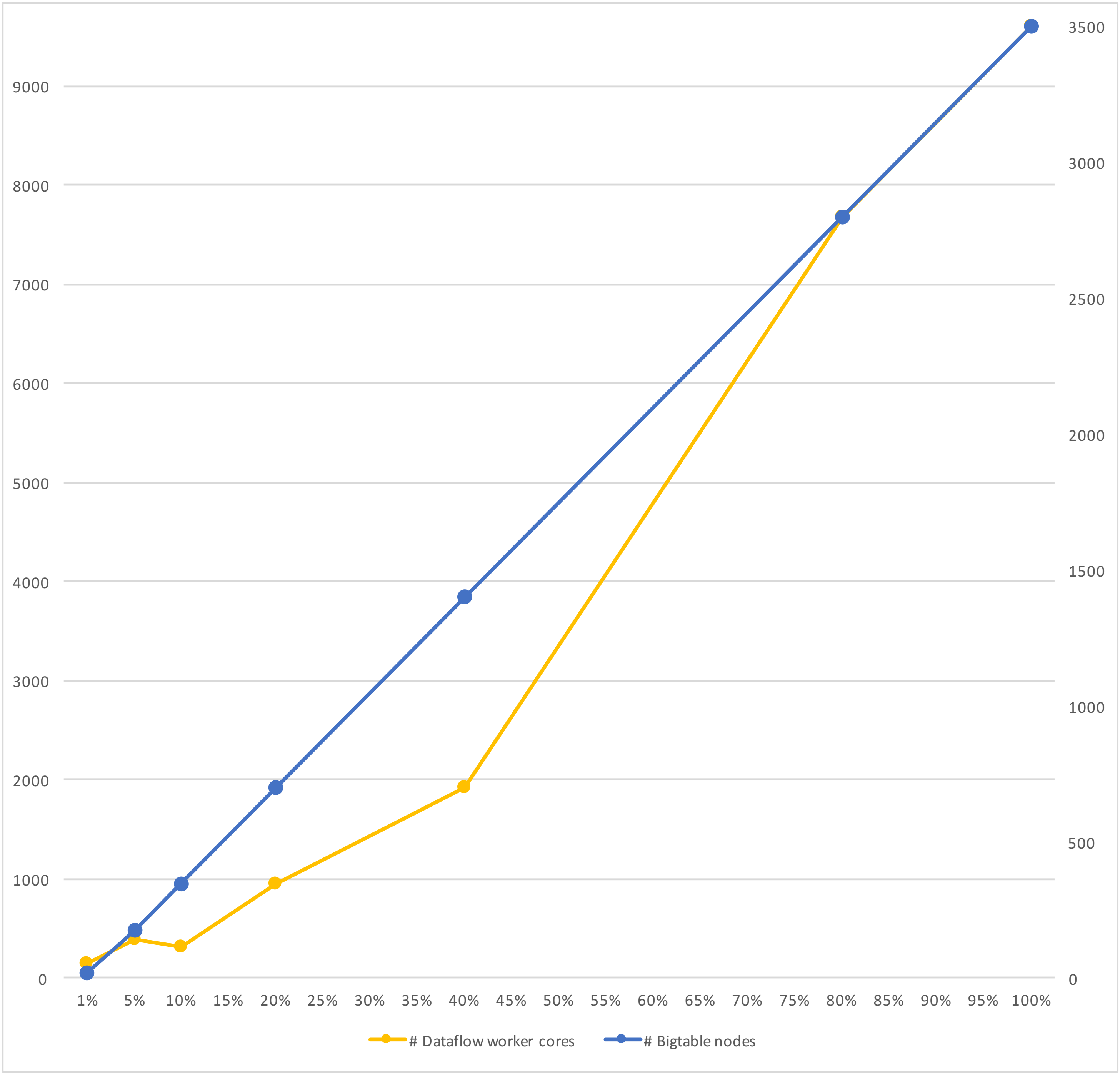


Figure 3: Dataflow Worker Cores (Left) vs. Bigtable Nodes (Right) – Dataflow Worker Cores from 8,680 to 9,600 Overlap with Bigtable Nodes from 2,800 to 3,500 (Source: Table 2)

3.2.3 Total execution time in minutes per target workload

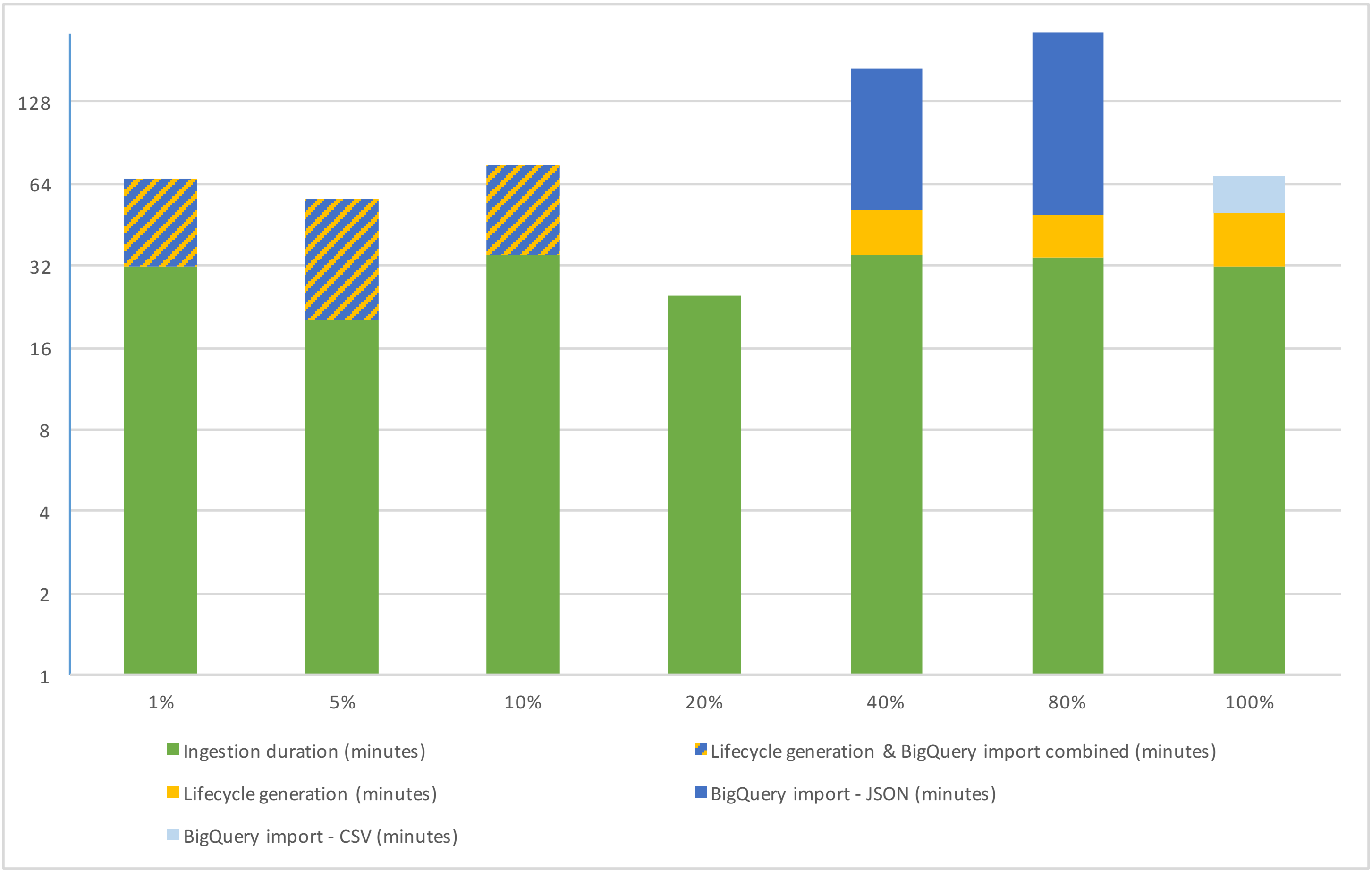


Figure 4: Stage Processing Time in Minutes (Vertical Axis, Base-2 Logarithmic) of the Target Message Amount (Horizontal Axis) (Source: Table 2)

4 Discussion

4.1 Key findings

Our observations indicate that, when employed interdependently, the various services of the GCP yield a solution architecture that scales nearly linearly.

While employing GCP as a computing provider relieves the infrastructure management burden from application and operations teams, data processing at this scale still demands a measured approach in order to realize maximum benefit. For applications processing such massive volumes of data, miniscule inefficiencies that might be ignored without consequence at smaller scales can, in fact, have considerable performance implications. Therefore, in order to identify and forestall potential workload bottlenecks, it is crucial to produce thoroughly-tested and highly-optimized code, while continually measuring key performance metrics across all architectural tiers.

4.2 GCP quotas

Despite cloud computing's promise of virtually unlimited, on-demand computing resources, by default, GCP does impose a limit to a project's immediately addressable computing resources. Since quotas are configured on a per-project and per-service basis, conducting a large-scale, end-to-end exercise such as this requires the coordination of multiple grants of quota escalation platform-wide. In many instances, a quota insufficiency in one service area manifests itself as a side effect within a separate service area. A common example is breaching an underlying Google Compute Engine memory, core or disk quota limitation, despite Dataflow worker counts being well within the default worker and job quota thresholds. In planning the execution of such workloads, it is helpful for teams to forecast their consumption of underlying resources, compare that forecast to a project's existing quota levels, and request threshold limit adjustments from Google accordingly.

For the architecture described herein, the specific quotas adjusted from GCP project defaults were:

- Compute Engine resources (cores, memory, IP, attached storage)
- Cloud Bigtable cluster size
- GCP limits as described in section 3.1
- Individual API quotas[43]

For API quotas specifically, using a large number of smaller workers for a particular pipeline can lead to more rapid starvation of available resources, since each worker node imposes its own API footprint when communicating with the Cloud Dataflow controller process. In the event that an API quota threshold is breached repeatedly, it may be beneficial to increase worker throughput by employing a smaller

43 "Dataflow API" from the Google Cloud Platform console – https://console.cloud.google.com/apis/api/dataflow.googleapis.com/usage

number of larger workers (i.e. scale vertically). Most quota breaches experienced during these exercises have in fact been of this "second order" variety. Thus, while linear application response is critical to a solution's ability to scale horizontally, ignoring the vertical dimension to sizing Dataflow pipelines could result in missed opportunities for valuable optimizations.

Quotas are typically granted to GCP services on a per-project, per-region basis.[44] Dataflow jobs, Bigtable clusters and Cloud Storage buckets can also be specified on a per-region basis. When managing quota limits, distributing job workloads according to region can be helpful in managing a project's resource expenditure.[45]

4.3 Worker ramp-up and tear-down time

When employing large numbers of Dataflow workers, ramp-up latency can be considerable as the Dataflow resource coordinator adds worker nodes in batches in response to escalating demand. In order to reduce this autoscaling latency, and because the specific workload goal of each iteration was known in advance, the team developed a tool to preempt the instantiation of pipeline worker nodes,[46] allowing the actual Dataflow pipeline activity to start in response to a Cloud Pub/Sub message instead. This can be viewed as starting the performance test with a warm cluster, since it allows processing at full throttle for the entire pipeline duration. When all workers that have been allocated for a particular job are ready, an out-of-band message is sent, received by a `PubSubIO` instance and the pipeline execution begins, capitalizing on all available workers.

Equally, Dataflow jobs do not report completion until all associated pipeline workers have been shut down. However, the shutdown process for workers begins when there are no more active `ParDos` within the pipeline, and does not wait for any downstream I/O operations to complete. Thus, projects are not billed for non-useful Compute Engine time even *within* an otherwise active pipeline. Developers must be circumspect with regards to pipeline architecture when deploying within a metered cloud environment, as any inefficiencies translate directly into the ultimate cost of computing the workload.

4.4 Worker machine type sizing

The sizing of the workers is a crucial characteristic of the Dataflow pipeline as it directly impacts Bigtable's throughput capabilities. Too few nodes will not allow the workload to extract Bigtable's maximum throughput yield. Too many nodes, on the other hand, can result in excessive timeouts and retries that can significantly contribute to throughput degradation.

As evidenced by our results, at smaller workload sizes, n1-standard-8 instances scaled better – a characteristic shared by earlier test results.[47] At a larger scale, n1-standard-32 instances yielded the best performance. This is a result of Bigtable's incremental network footprint as well as the footprint imposed by Dataflow job coordination.

4.5 Microbenchmarks

Given the sheer amount of processing power required for big data jobs, writing solid, optimized code is an absolute necessity. While we kept the source code underlying the test to strictly idiomatic Java, nevertheless, we adhered to the following performance optimization practices:

- Eliminate debug logging and/or otherwise arbitrary writes to `System.out`
    - In the case of exception logging, framework-specific methods should be used. Cloud Dataflow uses custom bindings to emit log events of all the common logging frameworks[48] to Cloud Logging. But if a stack trace is logged to STDOUT, this will cause a single log event per stack element/line in STDOUT, which is also very expensive.
- Cache local variables and derived computations, where possible
- Use `StringBuilder` in lieu of direct concatenation, where necessary
- Use Java 8's `java.time.*` package rather than `java.util.Date` and its cohorts
- Limit the use of Java's native exception model[49]
- Capitalize upon the Dataflow SDK's use of the template method pattern and organized our initialization routines appropriately.[50] We used `startBundle()` and `finishBundle()` to initialize and destroy resources. Invocations of `processElement()` should not be used to perform initialization duties for any dependent logic embedded within.

We found that the Java Microbenchmark Harness (JMH) was a helpful tool for reliable microbenchmarking during regression tests of both the functional and the non-functional requirements.[51]

4.6 BigQuery import

The implementation of the Dataflow SDK BigQuery adapter originally used for the test employed Javascript Object Notation (JSON) as its exchange format. Compared to other protocols, JSON is relatively verbose as each representative object is self-describing, leading to a large amount of data going over the wire being redundant. Furthermore, the additional markup with every record results in additional processing time as the import process must read in (then discard) the additional text. Since the object schema for each record was highly structured and uniform record-to-record, there was no particular upside to the JSON encoding for our specific use case.

Additionally, given the sheer amount of data being imported, fine-tuning was required to put a ceiling on the number of files that the Dataflow SDK BigQuery adapter had to address. If the number of temporary JSON files created by `BigQueryIO` exceeds 10,000, then the entire BigQuery import can fail.

44 "Resource Quotas" within the Google Cloud Platform – https://cloud.google.com/compute/docs/resource-quotas
45 The flip-side, of course, is that buckets and workers that are regionally co-located exhibit superior performance
46 PubSubStarter is available from https://github.com/SunGard-Labs/dataflow-whitepaper
47 Palmer, et. al (2015) *supra*
48 "Adding Log Messages to Your Pipeline" – https://cloud.google.com/dataflow/pipelines/logging
49 One third-party library employed for the test extensively used the exception class hierarchy and control flow during initialization. The performance implications were severe enough that our team resorted to overriding the library's fillStackTrace() method to ameliorate some of the performance impact, as more invasive adjustments to the library were impractical.
50 https://en.wikipedia.org/wiki/Template_method_pattern
51 JMH – http://openjdk.java.net/projects/code-tools/jmh/

Hence, for the 100 percent load test iteration, we authored a custom BigQuery import function that generates a CSV file per worker and saves the files (using `TextIO` from the standard Dataflow SDK) to Google's cloud storage service. This approach allowed us to reach the desired target execution time and also resulted in a very simple implementation. Since the FIX tags being persisted to BigQuery were known and static, no additional third-party libraries were used, which reduced any uncertainties related to performance characteristics under the current scenario. Only simple operations using `StringBuilder` and `byte[]` were necessary – again, backed by JMH tests to measure relative performance impact on each iteration.[52]

This is an excellent example of the seemingly small implementation details, often going unnoticed for smaller workloads, that can result in a material performance footprint when accumulated over millions of transactions per second.

4.7 Bigtable cluster distribution and scaling

By default, Bigtable analyzes table keyspaces and automatically compacts and rebalances data across cluster nodes. However, given the specific scale and workload performance that was required as part of this effort, it was decided to explicitly specify how Bigtable should distribute table data physically. This can help forestall unnecessary compaction operations and optimize the distribution and transfer of table data within the cluster as it scales up. This can be thought of as similar to the practice of explicitly specifying a large fixed heap size and/or garbage collection parameters to Java virtual machine (JVM) processes, to optimize the JVM's time spent on heap maintenance.

Each market event message is tagged with a UUIDv4 value that is randomly and uniformly distributed over the entire keyspace. Because of this distribution profile, it was decided to explicitly configure the splits at the time of table definition. To avoid performance degradation stemming from the "hot" node phenomenon that can plague tables with poor keyspace distribution, Bigtable will periodically evaluate the key space distribution of the table. As UUIDv4 values are, in general, random and therefore exhibit broad distribution over the entire keyspace, it is nearly impossible for Bigtable to glean a legitimate pattern from which to make rebalancing decisions, especially when the table contains only a small percentage of its ultimate volume. This could lead to a situation in which Bigtable assumes a row key, such as "`123e4567-e89b-12d3-a456-426655440000`," as the upper boundary of the keyspace – which is not necessarily correct based upon the entire ID space. Before data is ingested, the keyspace distribution is already known (which may not be the case in other circumstances), so it is sensible to direct Bigtable accordingly.

```
CREATE 'myTable', {NAME='CF1'}, {NAME='CF2'},
SPLITS => ['keyboundary1', … , 'keyboundaryN']
```

Based on our previous tests with smaller datasets and running the same data set using multiple factors, we found that a factor of 24 times the amount of Bigtable nodes provided optimal throughput. In other words, for a Bigtable cluster with 10 nodes, 240 splits are used during table creation.[53]

The process behind arriving at this factor is lengthy, but follows a distinct pattern. The Dataflow worker nodes must be sufficiently sized such that they themselves are not the limiting factor. When this has been achieved, a complete run is conducted while monitoring Bigtable's write throughput performance via the Google Cloud Console. If the observed write performance experiences significant deterioration after its initial increase, yet ultimately reaches its final write performance later on, the deterioration is quite likely due to Bigtable's rebalancing. Eventually, the split initially chosen becomes less optimal as the totality of the dataset (and its associated keyspace) converges.

At this point, iterations of adjusting the split count then re-running the test (while keeping all other factors stable) should improve the observed throughput.

This process is repeated until no further deterioration is observed. One aspect to validate during this exercise is that the increased Bigtable performance does not encounter additional bottlenecks elsewhere within the solution.[54]

4.8 Using Bigtable for stateful pipelines

The ability of application logic to scale linearly based upon workload demand is inextricably linked to the logic's reliance upon a shared state. The facilities and API semantics of the Dataflow SDK abstract the majority of this complexity from the software engineer. This is not to say that developers are inoculated against all pitfalls; however, with no way to enforce shared-state restrictions at compile time, there remains ample opportunity for developers to introduce inefficiencies.[55] In order to dispatch logic upon the inbound datasets with near-linear scaling performance, all information required by the `processElement()` method must be known at the time of entry. Dependency upon any external, mutable and shared data source introduces non-functional hazards into the system, such as performance-degrading synchronization points.

As such, the linkage algorithm conducts its own message processing based upon immutable input values and ensures that, over time, all requisite information falls into the right place – not unlike the game Tetris® where each element contributes to the greater structure by being placed in the right position upon arrival. As further elements are added to the structure, ultimately the final superstructure (pipeline) representation converges.

One area where Bigtable has helped in this regard is in supporting row inserts from multiple, parallel PUT operations, with the final row represented by the cumulative information of all preceding `PUT`s.

52 See 4.5 Microbenchmarks
53 The Split Generator source code by FIS Global can be found at https://github.com/SunGard-Labs/dataflow-whitepaper/blob/master/src/main/java/com/sungard/dataflow/BigtableSplitGenerator.java
54 See 4.9 Google Cloud Platform vs. private infrastructure
55 Sferrazza, et al (2015)

4.9 Google Cloud Platform vs. private infrastructure

Cloud computing offers unprecedented power to consumers without the burden of traditional, large upfront capital investment in infrastructure and support. However, for workloads operating at terabyte or petabyte processing scales, a circumspect and evidence-based approach to deriving the ultimate solution architecture is necessary, to which in-house approaches should also subject themselves. Applying small, incremental changes that evolve the solution in small steps is preferable. While stringently and relentlessly measuring the impact of each change can often be tedious, this practice enabled our team to navigate the associated architectural and engineering trade-offs with confidence.

Given the isolation of individual cloud services, they can simply be added to the system on-demand – without an associated, up-front capital investment. In addition to the financial advantages, this poses great benefits to developers as well, as it removes barriers to experimentation and potentially reduces time-to-insight.[56] Different services may be added provisionally to evaluate their suitability for a particular function, and evidence of this suitability may be collected to assist in arriving at an architectural decision. One does not need to conduct a lengthy procurement cycle only to learn ultimately that the service does not meet expectations.

This methodology is especially important as components are added to an evolving solution architecture, since the isolation of individual factors affecting system performance becomes more challenging and time-consuming. Therefore, our typical approach was to conduct initial tests with abridged datasets while constantly measuring key performance metrics of the particular system-under-test against any individual configuration profile.

As bottlenecks emerge in any one component, improvement efforts should be focused there. However, it is important to note that the "component" in question might prove to be any combination of custom application code, external resources or even quotas imposed by the underlying computing provider.[57] Once a bottleneck is removed, the throughput of the system is free to scale,[58] and the infrastructure underpinning the solution can dynamically and predictably align to end-user demand.

Additionally, the segregation of each cloud service as a discrete component facilitates performance tuning overall. When bottlenecks emerged within any particular service, the universe of possibilities for optimization were restricted to the specific service's configuration parameters. While this limitation may be a departure for engineers who are comfortable crossing multiple OSI layers during troubleshooting exercises, limiting the tuning parameters to a service-specific configuration presented a clear productivity benefit for our development team.

One practical aspect of conducting such large-scale testing exercises of which teams must be mindful is the decommissioning of resources between active testing iterations. While the volumes conducted in this test are exaggerated for many typical workflows, even lower-demanding testing iterations can consume a large amount of compute, I/O, service and storage resources. Hence, automating the decommissioning of resources as part of the test iteration regimen is essential for the precise quantification of workload cost as well as for avoiding expenditure on unproductive resources.

5 Conclusion

Based upon our findings and the overall insights gained,[59] the approach of relentlessly capturing performance metrics while increasing workloads incrementally resulted in a very positive testing outcome. As always, system readiness tests must be budgeted from both a human and computing resource standpoint. The specific iterations required to conduct massive scaling exercises entail the extensive coordination of both systems and personnel. Thus, it is wise to plan the effort carefully, and focus on maximizing the system's available throughput across every dimension of the overall solution architecture.

Mitigating the computing burn rate can be achieved by optimizing application code and algorithms, and avoiding computationally expensive operations when there may be simpler alternatives, such as `System.arraycopy()`.[60] It is also worth mentioning that each major Java version has significantly different execution behaviors with regards to the runtime's component base classes (`String, java.math`, wrapper types, etc.). Therefore, it is helpful to conduct microbenchmarks in tandem with the application of software engineering best practices, in order to build evidence to support or refute assumptions and expectations about code paths. Additionally, the refactoring of application code often goes hand in hand with performance improvements, further incentivizing and rewarding optimizations at this scale.

The escalation of data quantity and velocity, in no way restricted to the financial space, portends that a move to cloud computing is desirable and inevitable across many industry sectors. In order for engineering teams to extract the most benefit from these capabilities, application development approaches must evolve accordingly. Dataflow, Bigtable and BigQuery liberate developers from most complexities underlying big data application development and infrastructure deployments for distributed, parallel systems. They allow programmers and engineers to operate on a higher level, one much closer to the specific problem domain being addressed. While this combination of services significantly raises the bar on what even modestly-sized engineering teams can accomplish in a short duration, extracting the maximum symbiotic benefit undoubtedly requires adjustment from traditional development approaches.

56 "How Time-to-Insight Is Driving Big Data Business Investment" http://sloanreview.mit.edu/article/how-time-to-insight-is-driving-big-data-business-investment/
57 See 4.2 GCP quotas
58 Goldratt (1984)
59 See 4 Discussion
60 System's arraycopy() is a native method that typically provides superior performance for operations on arrays

6 Acknowledgements

We would like to thank the following people for providing their help, support, and technical and domain expertise in building this next generation prototype: Todd Ricker, Benjamin Tortorelli, Seong Kim, Meghadri Ghosh, Chris Tancsa, Petra Kass, Regina Brab and Marianne Brown at FIS, and especially Carter Page, Solomon Duskis, Misha Brukman, Phaneendhar Vemuru, Dan Halperin, William Vambenepe and the rest of the Google Cloud Bigtable and Dataflow teams, who have been awesome (again).

7 References

Rule 613 (Consolidated Audit Trail) https://www.sec.gov/divisions/marketreg/rule613-info.htm

N. Palmer, Y. Wang, M. Sherman, S. Just (2015) *"Scaling to Build the Consolidated Audit Trail: A Financial Services Application of Bigtable"* https://cloud.google.com/bigtable/pdf/FISConsolidatedAuditTrail.pdf

J. Dean (2008) *"Software Engineering Advice from Building Large-Scale Distributed Systems"*

M. Öhman (2013) *"A Data-Warehouse Solution for OMS Data Management"*

S. Sferrazza, S. Just (2015) *"Transforming Options Market Data with the Dataflow SDK"* https://cloud.google.com/dataflow/pdf/TransformingOptionsMarketData.pdf

E. Goldratt (1984) *"The Goal: A Process of Ongoing Improvement"*

J. Spolsky (2002) *"The Law of Leaky Abstractions"* http://www.joelonsoftware.com/articles/LeakyAbstractions.html

F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber (2006) *"Bigtable: A Distributed Storage System for Structured Data"* http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

N. Palmer, J. Chen, S. Sinha, F. Araujo, M. Grinthal, F. Rafi (2016) "*Market Reconstruction 2.0: Visualization at Scale*"