Performance and Scalability with Griddable.io

Executive summary

Griddable.io is an industry-leading timeline-consistent synchronized data integration grid across a range of source and target data systems. Griddable.io's flexible shared-nothing architecture with strong consistency guarantees allows for building highly available and scalable deployments. In this whitepaper, we describe the performance characteristics based on many controlled tests that we have executed. We show how the grid can be scaled to achieve throughput of hundreds of thousands of events and hundreds of megabytes per second in both its relay and consumer tiers.

Introduction

Griddable.io is a SaaS company that delivers a platform for synchronized data integration across enterprises and clouds.

Griddable.io provides:

- Source-independent transactional grid optimized for change data based replication
- Policy engine for selective replication, on-the-fly data transformation and conflict resolution
- Pluggable architecture to extend support for data sources, consumers, and policies
- A set of tools and web-based UI to simplify the management of the grid in a geo-distributed environment

Scalability is one of the most important aspects of our data synchronization grid. There are multiple dimensions to scalability.

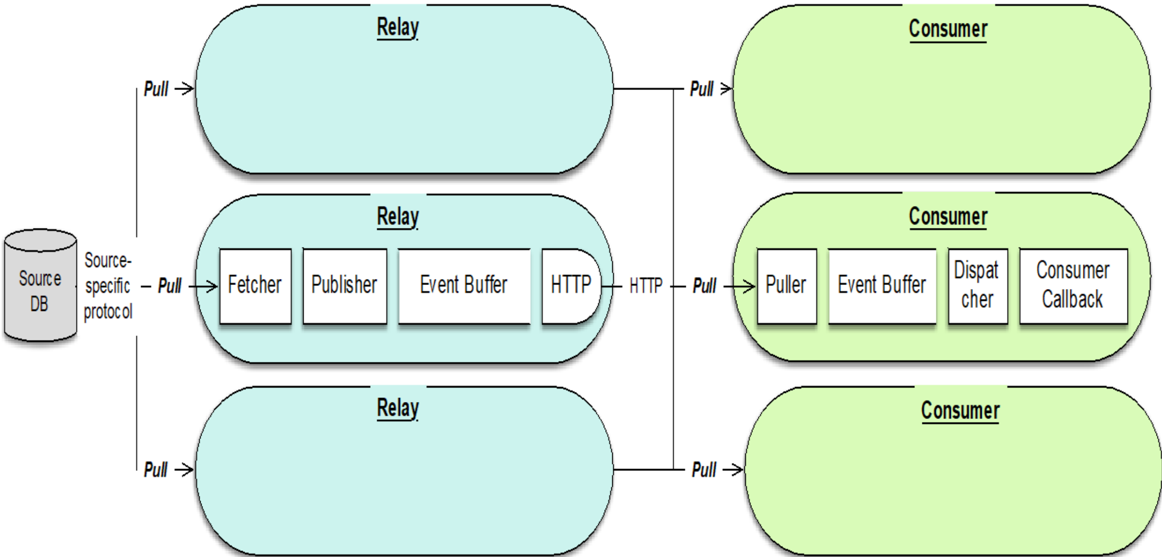

Figure 1: Griddable.io's real-time data synchronization path

One dimension is how fast each individual component can read and process the event stream. Another important aspect of scalability is how to scale the system beyond a single component. In this paper, we'll focus on the real-time replication path from the source database through the relay to the consumers

Furthermore, the relay tier has an inbound throughput aspect (how quickly it can read from the upstream database) and an outbound throughput aspect (how quickly it can serve the consumers).

In this paper, we'll study the throughput in terms of events per second and megabytes per second of both single instances and clusters of relays and consumers. We study those in the context of two popular source databases – Oracle using our GoldenGate fetcher and MySQL using our binlog fetcher.

The rest of whitepaper is structured as follows. First, we provide a quick overview of Griddable.io's architecture. Then, we present our performance results for individual components and show how we can scale those numbers by adding instances and partitioning the traffic. Next, we discuss future work on additional performance experiment and we present features that we are working on that can further improve performance and scalability. Finally, we summarize our findings and present conclusions.

Architecture

In this paper, we study the real-time data synchronization scalability and performance of Griddable.io's grid. The real-time path consists of the following systems (see Figure 1):

- Source database whose changes are captured and fetched by the relay
- Relay that is responsible for capturing the changes from the source database and publishing them to the grid
- Consumers that subscribe to the change stream for a specific database

The relay *Fetcher* pulls the change data using a source-dependent protocol. For example, this is JDBC for Oracle (using the LogMiner fetcher), or GoldenGate Java exit for multiple databases including Oracle, or the binlog protocol for MySQL and MariaDB. The database change events are converted into source-independent replication events and replication policies are applied to these events. The events are then published transactionally by the relay *Publisher* to the relay *Event Buffer* where they are available for consumption by downstream subscribers, the consumers.

The consumers continuously pull replication events from the relay over HTTP protocol. The consumer *Puller* stores those events in a consumer *Event Buffer* while there is free space in the buffer. The consumer *Dispatcher* asynchronously pulls events from the event buffer and passes them to the *Consumer Callback* for processing. Once the consumer callback processes all events in a transaction, those events are removed from the event buffer to free up space for the Puller to fetch more events from the relay.

The entire real-time data synchronization path forms an asynchronous multi-staged processing pipeline designed for high throughput and low latency.

To further improve scalability, replication and partitioning can be employed:

- Relay instances can be replicated so that they can support more consumers
- Relay instances can be partitioned (though fetcher policies) so that each instance processes a portion of the incoming database traffic. On aggregate the partitioned relays can process higher volumes of incoming data.
- Like the relays, the consumer instances can be partitioned so that each consumer processes only a portion of the replication events stream.
- The performance configurations in this paper employ all the above techniques to maximize throughput for the lowest latency.

Environment and workload

We measured the effective throughput of change-data-capture events through the grid in various stages of the pipeline across several different hardware configurations.

Environment

All the tests performed have been conducted in the cloud, on Google Compute Engine, by provisioning several VMs to host our services. During these tests, we will perform the same test across several machine-types to show how the architecture takes advantage of increased capacity on the same VM. The machine-types we use are the following:

Machine Type	Description
N1-standard-2	Standard machine type with 2 virtual CPUs and 7.5 GB of memory.
N1-standard-4	Standard machine type with 4 virtual CPUs and 15 GB of memory.
N1-standard-8	Standard machine type with 8 virtual CPUs and 30 GB of memory.

We have configured these VM instances with SSD and they are running Linux Ubuntu Xenial 16.04. To ensure minimal network latency, we have provisioned each of the VM instances in the same region and availability zone.

A full specification of these machine types can be found here:

https://cloud.google.com/compute/docs/machine-types

Workload

For each of the tests conducted, we will configure a source database and create a workload of database transactions that will generate redo events to mine as part of replication.

For the purposes of this test, we have chosen MySQL 5.7 as the source database. The load consists of 4,800,000 record updates to the database with no replication active at the time, such that the entire load will be available once the test is started. In MySQL, we use *binlog* as the source of change-data events. More information on binlog can be found at:

https://dev.mysql.com/doc/refman/5.7/en/binary-log.html

The content of the change data consists of 3 tables, with the size of each record in the table about 1100 bytes. As the load is generated, each of the columns are filled with random data. The load is evenly spread amongst the 3 tables.

MySQL is running on its own machine on a 16-CPU VM to ensure it can handle the load.

Methodology

A typical Griddable.io pipeline consists of the following:

1) A source database
2) One or more relays
3) One or more consumers
4) A target destination (DB, KV store, FS, etc.)

In the following diagram, we illustrate the various stages in a simple pipeline. Due to the flexible topology inherent in our architecture, more advanced configurations are possible through chaining, but for this test, the following flow was used.

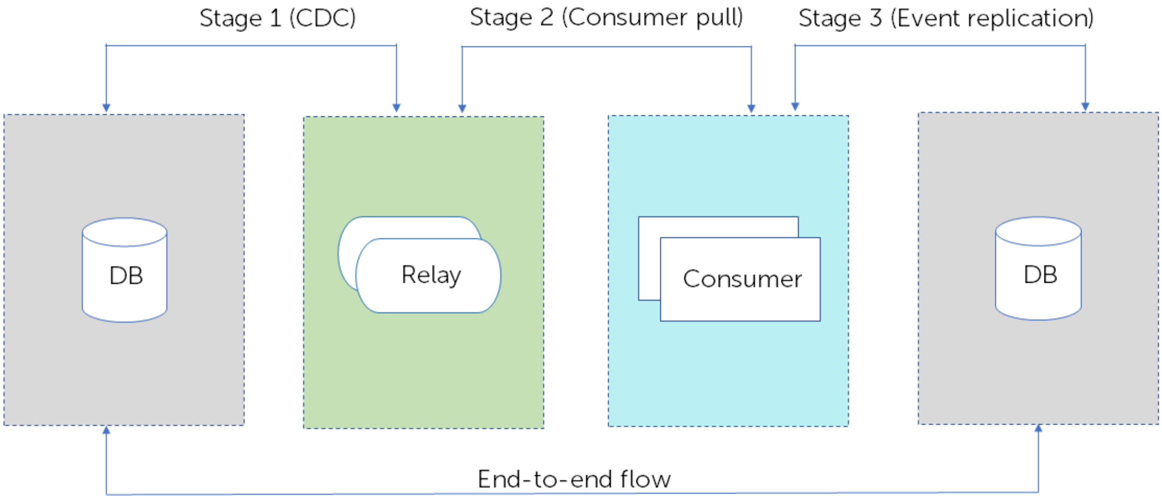

Figure 2: Performance methodology

In our testing, we will measure Stage 1, Stage 2, and end-to-end throughput. The final stage has been mocked out in this test, due to the additional cost and complexity in configuring databases that can sustain a write workload of hundreds of thousands of database records per second. Even without that final stage, we can accurately measure the performance of the grid as change-data flows through it.

Scalability

Measuring Stage 1 performance (MySQL Binlog to relay)

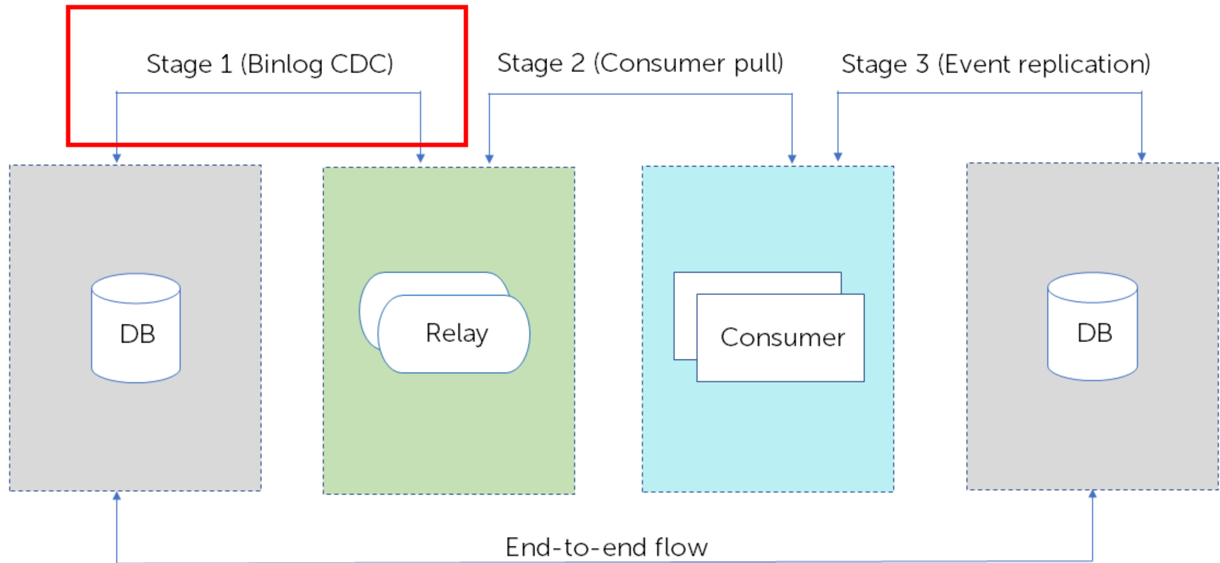

Figure 3: Stage 1 performance

To measure Stage 1 performance, we use a load generator that creates database updates that in turn generates replication log entries. While this load generation occurs, neither the relay services nor the consumer services are running. Once the load generation is complete, the Relay(s) are started up and they begin mining change data events from the replication log via MySQL binlog, and timing measurements are written through logging to capture the beginning timestamp and the ending timestamp for the load. In this case, since the consumer is not running the data will stop flowing at the end of Stage 1. The following table lists the measured performance across several different configurations.

Table 1: VM configurations and results from Stage 1

Instance Type	VMs	Configuration	# of records	record size	CPU load	vCPUs	duration	rate ev/s	rate Mbit/s
n1-standard-2	1	single relay 2-cpu	4,800,000	1100	0.90	1.80	156.10	30,750	338
n1-standard-4	1	single relay 4-cpu	4,800,000	1100	0.78	3.10	112.78	42,560	468
n1-standard-8	1	single relay 8-cpu	4,800,000	1100	0.62	4.97	62.10	77,295	850
n1-standard-8 x 2	2	2 relays 8-cpu	4,800,000	1100	0.56	8.88	42.44	113,114	1,244
n1-standard-8 x 3	3	3 relays 8-cpu	4,800,000	1100	0.55	13.15	29.20	164,367	1,808

The first 3 entries in this table show throughput scaling vertically on the same VM instance as the number of CPUs are increased. In the 4th entry, we scale horizontally by adding a second relay on another VM instance. In this case, the 4,800,000 records in the load are distributed to the 2 relays via our policy filtering mechanism. In this example, we filter the incoming change data events such that separate tables are handled by separate relays. The CPU measurements and rates are aggregated

across the two instances. Similarly, in the final entry in the table we distribute the load across 3 relays resulting in 164,367 events per second and a data throughput rate of 1.8Gbps. Additional gains can be realized by further partitioning the load and adding more relays. The following graphs depict the resulting data.

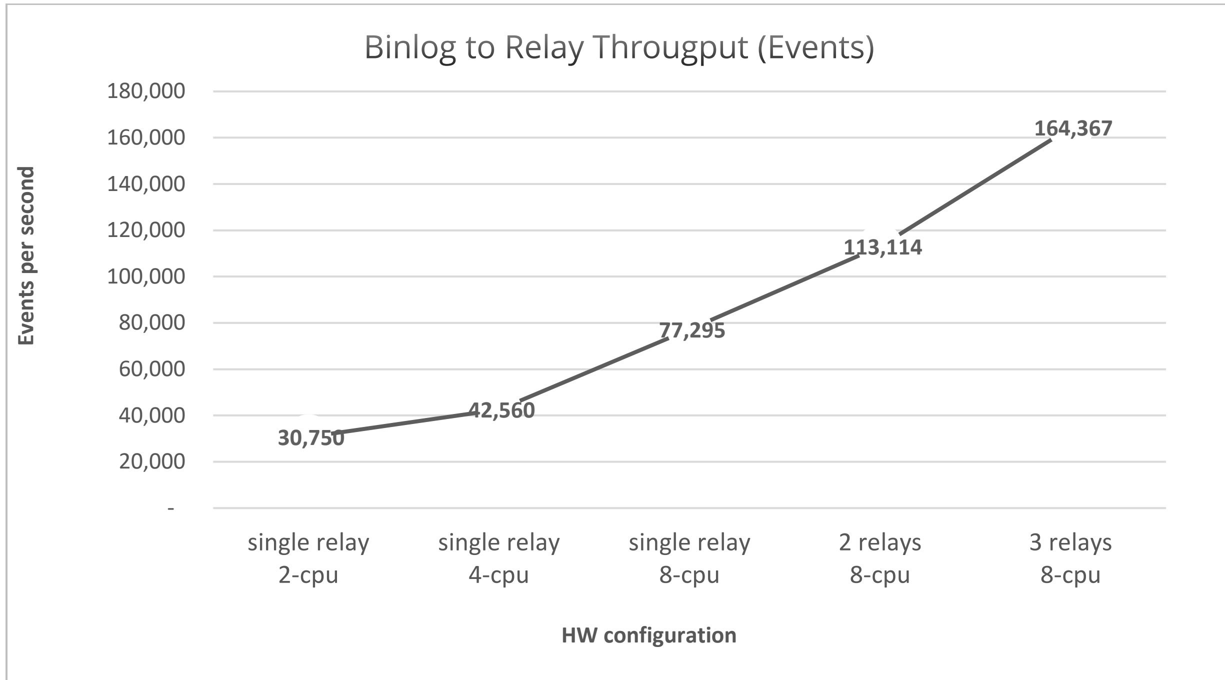

Figure 4: Binlog to relay throughput

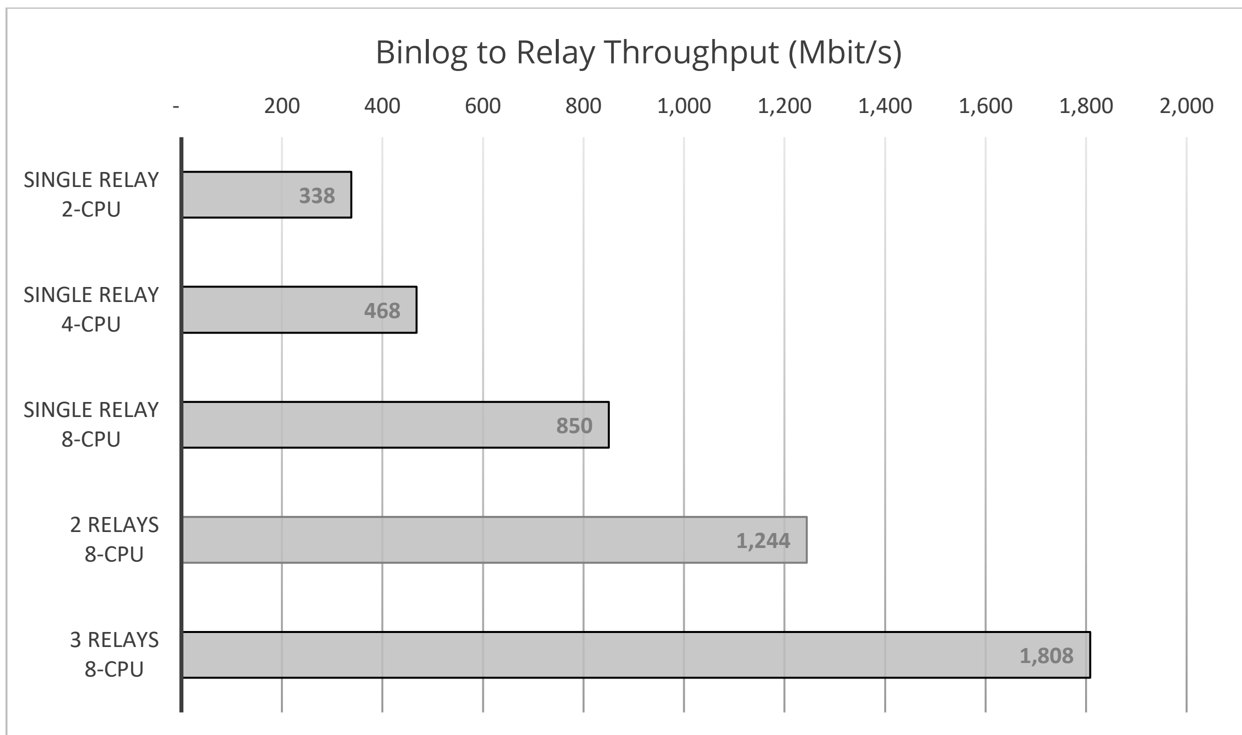

Figure 5: Binlog to relay throughput

Measuring Stage 2 performance (Relay to consumer)

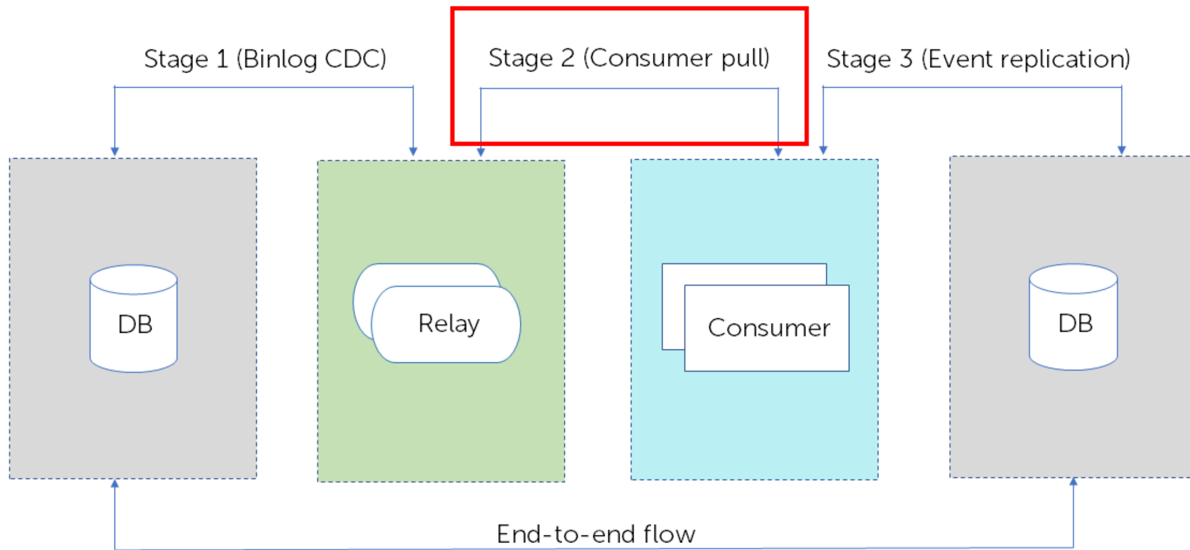

Figure 6: Stage 2 performance

To measure Stage 2 performance, the change data is queued in the relay waiting for the consumer(s) to pull the events. Since Griddable.io's grid is designed as a streaming data pipeline, we had to configure the relay with a larger in-memory buffer to queue the change-data. For this test, we provisioned a single 8-way 30GB memory VM instance for the relay and then varied the configuration of the consumer(s).

First, the load generator generated the 4.8M records that were preloaded into the relay event buffer. Once the records were fully queued, we started up the consumer(s) with timestamped logging in the consumer's logfiles to track the length of time that elapsed from the first event to the last event. The following table shows the results.

Table 2: VM Configurations and Results for Stage 2

Instance Type	VMs	Configuration	# of records	record size	CPU load	vCPUs	duration	rate ev/s	rate Mbit/s
n1-standard-2	1	single consumer 2 cpu	4,800,000	1100	0.95	1.90	110.08	43,605	480
n1-standard-4	1	single consumer 4 cpu	4,800,000	1100	0.54	2.17	78.93	60,815	669
n1-standard-8	1	2 consumers 8 cpu	4,800,000	1100	0.59	4.72	52.88	90,765	998
n1-standard-8	1	3 consumers 8 cpu	4,800,000	1100	0.79	6.32	35.72	134,378	1,478

In the first two tests, we ran a single consumer across two different machine-types on Google Compute Engine with an increase in performance relative to the number of CPUs available. For the 3rd and 4th test, we increased the configuration to an 8-cpu machine and we partitioned the load to scale through multiple consumers. The following graphs depict the increase in throughput as we add additional capacity.

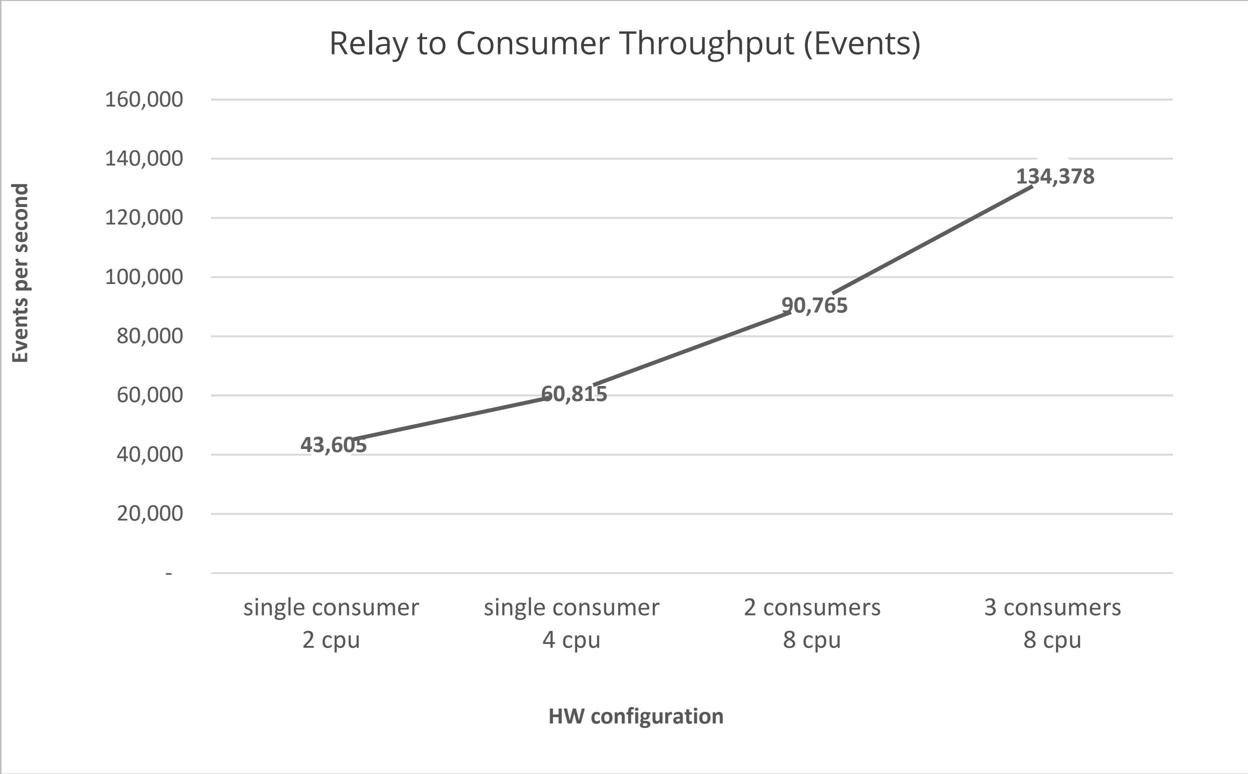

Figure 7: Stage 2 throughput

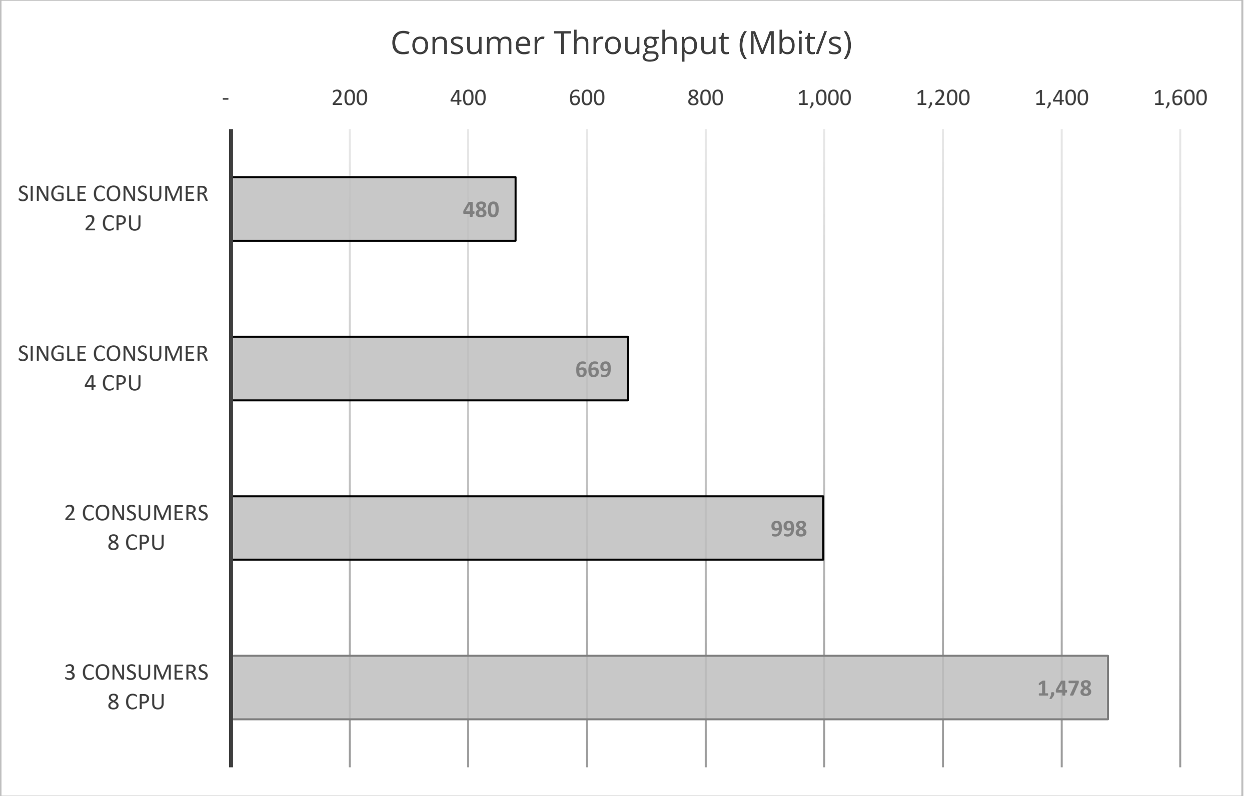

Figure 8: Consumer throughput

In these tests, all the consumers were run on the same VM however if network I/O or disk I/O became the bottleneck on a single machine, the consumers could be easily distributed across different machines.

End-to-end (MySQL binlog to mock database)

The final step of benchmarking is to measure the end-to-end replication performance (with a mocked-out target DB) from the MySQL binlog through the consumer. We only performed this run for the largest configuration.

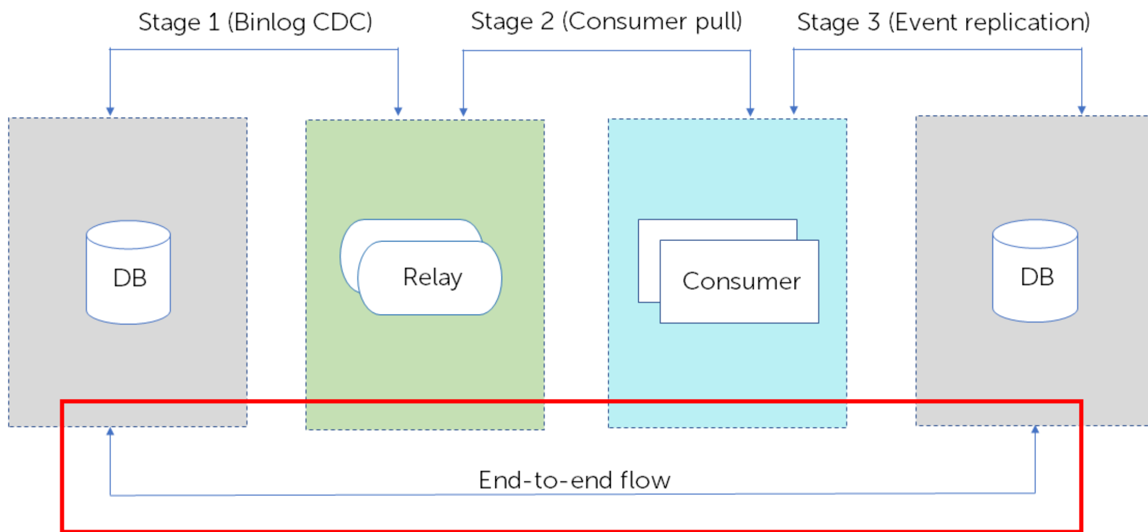

Figure 9: End-to-end MySQL binlog

Using the same load as the previous tests, we have created the following configuration to process the end-to-end load.

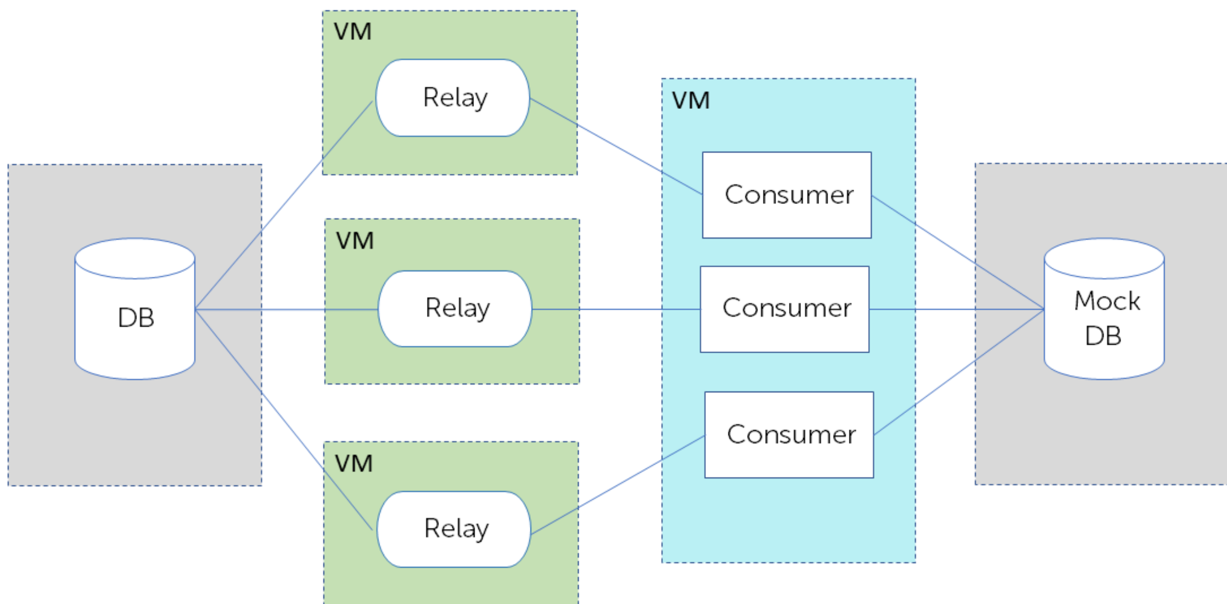

Figure 10: End-to-end setup

And the results are presented in the table below:

Table 3: End-to-end results

Instance Type	VMs	Configuration	# of records	record size	duration	rate ev/s	rate MBit/s
n1-standard-8	4	3 relays, 3 consumers, 8 cpu	4,800,000	1100	33.66	142,602	1,569

From our Stage 1 testing, we saw that the relays (3) could process incoming change data events at a rate of 164k events per second. During our Stage 2 testing, we saw that the consumers (3) could process at 134k events per second. When put together, they could stream from MySQL's binlog through the consumer to a mock DB at a rate of 142k events per second. The slight uptick in performance is likely due to consuming from 3 relays for this final test, rather than a single relay in the Stage 2 test.

Oracle GoldenGate performance

As an additional data point, we have measured replication performance with an Oracle GoldenGate-based relay fetcher. Although GoldenGate is already a replication product from Oracle, we have created a java-based Extract that plugs into GoldenGate which can process change-data events using GoldenGate as the source.

Our testing environment includes a pair of Google Compute Engine VM's (machine-type N1-standard-16). In this environment, our relay runs on the same VM as GoldenGate and Oracle. In this configuration, we are only running a single relay and a single consumer. As the previous section shows, if data can be partitioned across the pipeline, serious performance gains can be realized by scaling the system.

The target system is a VM that is running HBase, using a phoenix jdbc driver to access HBase. In this case, the change data is not replicated to the same schema, but all change-data is written to a single table that contains the change-data in it.

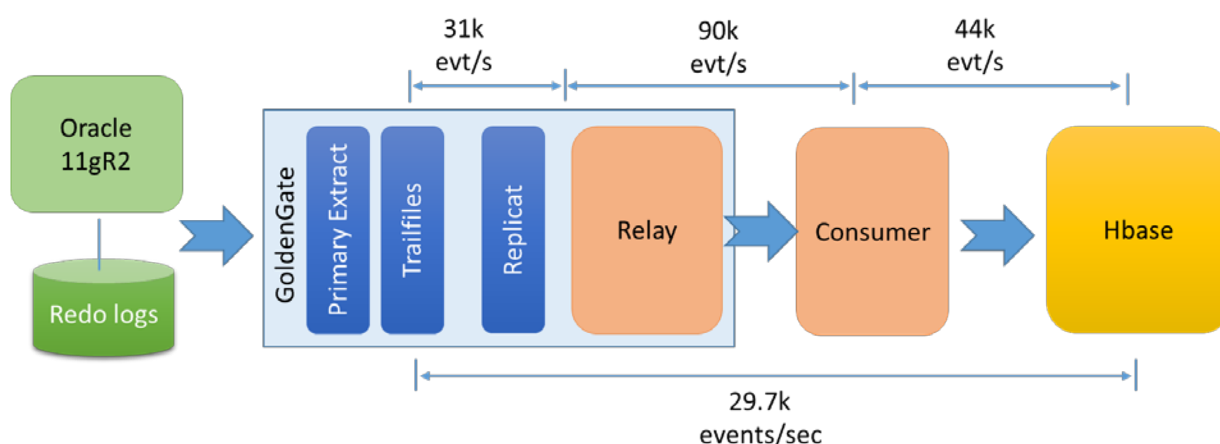

Figure 11: Oracle GoldenGate to HBase

The load generation setup is like the previous MySQL performance runs. We are writing to two tables on the source database with an approximate record size of 1100 bytes. The load consists of 2M

records and the same methodology was used to capture throughput measurements on the individual stages in the pipeline.

Future work

We plan to implement other improvements that will further raise the performance of Griddable.io's synchronized data integration grid. Our current consumer supports transaction batching (aggregating multiple transactions into one write). Transaction batching is critical for destinations where exactly-once is required, since it minimizes the number of round-trips to the target system (database) when the transactions are applied serially.

Automatic consumer load-balancing

Currently, partitioning of the replication event stream among a group of consumer instances requires manual specification of the policy for each consumer.

In our next release, we will provide automatic partitioning based on the Apache Helix[1] cluster management framework. Given a set of policies which define the available partitions, they will be automatically distributed among the available consumer instances. Checkpoints are shared and if one consumer instance becomes unavailable, the partitions for which this instance was responsible will be distributed among all other available instances.

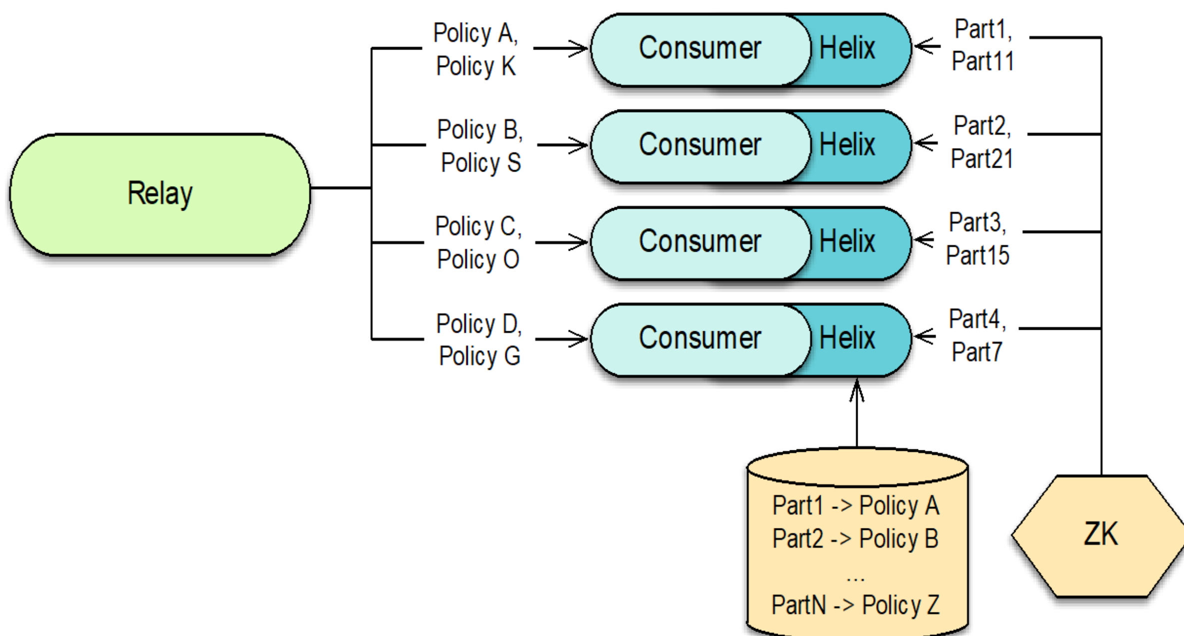

Figure 12: Automatic consumer load-balancing

This approach is particularly useful if the partitioning can be done based on schema or based on tenant. As an example, the object materialization consumer will greatly benefit from this partitioning

---

[1] helix.apache.org

when applied on a per-tenant basis. Scalable consumer clusters can be built while minimizing the operational overhead.

Consumer micro-batching

Currently, some consumer implementations (e.g., our Phoenix consumer) have an ability to perform an internal parallelization of the processing of events. This can be particularly useful for alleviating latencies associated with writing to remote data stores or systems.

The core idea is to partition the incoming transaction batch (a group of one or more transactions) into *micro-batches*. The partitioning is performed by a pluggable *Partitioner* component. Partitioning can be done on different criteria, from an event round-robin through a transaction round-robin to partitioning based on a field (e.g. customer ID).

Each micro-batch is processed and stored to the backend database (or other system) in parallel. A *Committer* component keeps track of which micro-batches have committed and stores that in an internal checkpoint. Once all micro-batches have been applied, the internal checkpoint is advanced.

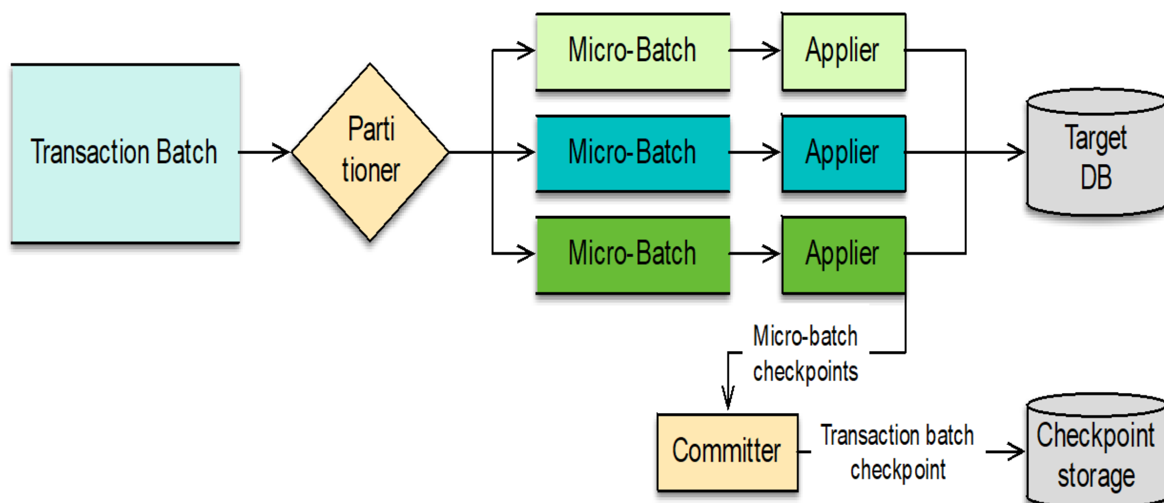

Figure 13: Consumer micro-batching

We are planning to generalize that approach as a common pattern available for all consumer implementations.

Combinations of multiple scaling techniques are also possible. Micro-batching can be combined with transaction batching and can also be automatically load-balanced based on schema or tenant.

Long-poll consumer protocol

Currently, the consumer polls the relay periodically for new events through HTTP request. Higher frequency of polling improves the replication latency to the consumer but increases the per-consumer load on the relay (HTTP request processing, increase garbage collection).

An important planned performance optimization is to establish a long-running request where the relay never terminates the response and new events in the relay's Event Buffer are sent immediately to all

subscribed consumers with applicable replication policies. This will allow consumers to run at optimal latency with minimal effect on relay CPU utilization.

On-the-wire compression

In cases, where events contain large amounts of data (especially, text data), compression can significantly improve bandwidth utilization. We plan to add on-the-wire compression for the events sent from the relay to the consumer. We expect the effectiveness of the compression to increase with the long-poll consumer protocol feature.

Conclusion

We have measured the throughput of Griddable.io's grid across various stages with multiple scaling configurations. We observed that each individual component of the system can scale to tens of thousands of events and tens of megabytes per second. Further, we have shown that with horizontal scaling we can increase the aggregate throughput to hundreds of thousands of events and hundreds of megabytes per second. The system scales almost linearly, with the addition of new relay and consumer instances, which indicates that there is no inherent software bottleneck.

griddable.io

2540 North First Street, Suite 201
San Jose CA 95131 USA
Phone 669.284.2143
www.griddable.io
© 2018 Griddable, Inc. All rights reserved. Griddable is a registered trademark of Griddable in the United States. All other company and product names may be trade names or trademarks