Procella: Unifying serving and analytical data at YouTube

Biswapesh Chattopadhyay Priyam Dutta Weiran Liu Ott Tinn
Andrew Mccormick Aniket Mokashi Paul Harvey Hector Gonzalez
David Lomax Sagar Mittal Roee Ebenstein Nikita Mikhaylin Hung-ching Lee
Xiaoyan Zhao Tony Xu Luis Perez Farhad Shahmohammadi Tran Bui
Neil McKay Selcuk Aya Vera Lychagina Brett Elliott
Google LLC
procella-paper@google.com

ABSTRACT

Large organizations like YouTube are dealing with exploding data volume and increasing demand for data driven applications. Broadly, these can be categorized as: reporting and dashboarding, embedded statistics in pages, time-series monitoring, and ad-hoc analysis. Typically, organizations build specialized infrastructure for each of these use cases. This, however, creates silos of data and processing, and results in a complex, expensive, and harder to maintain infrastructure.

At YouTube, we solved this problem by building a new SQL query engine – Procella. Procella implements a superset of capabilities required to address all of the four use cases above, with high scale and performance, in a single product. Today, Procella serves hundreds of billions of queries per day across all four workloads at YouTube and several other Google product areas.

PVLDB Reference Format:
Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Nikita Mikhaylin, Selcuk Aya, Vera Lychagina, Brett Elliott. Procella: Unifying serving and analytical data at YouTube. *PVLDB*, 12(12): 2022-2034, 2019.
DOI: https://doi.org/10.14778/3352063.3352121

1. INTRODUCTION

YouTube, as one of the world's most popular websites, generates trillions of new data items per day, based on billions of videos, hundreds of millions of creators and fans, billions of views, and billions of watch time hours. This data is used for generating reports for content creators, for monitoring our services health, serving embedded statistics such as video view counts on YouTube pages, and for ad-hoc analysis.

These workloads have different set of requirements:

- **Reporting and dashboarding:** Video creators, content owners, and various internal stakeholders at YouTube need access to detailed real time dashboards to understand how their videos and channels are performing. This requires an engine that supports executing tens of thousands of canned queries per second with low latency (tens of milliseconds), while queries may be using filters, aggregations, set operations and joins. The unique challenge here is that while our data volume is high (each data source often contains hundreds of billions of new rows per day), we require near real-time response time and access to fresh data.

- **Embedded statistics:** YouTube exposes many real-time statistics to users, such as likes or views of a video, resulting in simple but very high cardinality queries. These values are constantly changing, so the system must support millions of real-time updates concurrently with millions of low latency queries per second.

- **Monitoring:** Monitoring workloads share many properties with the dashboarding workload, such as relatively simple canned queries and need for fresh data. The query volume is often lower since monitoring is typically used internally by engineers. However, there is a need for additional data management functions, such as automatic downsampling and expiry of old data, and additional query features (for example, efficient approximation functions and additional time-series functions).

- **Ad-hoc analysis:** Various YouTube teams (data scientists, business analysts, product managers, engineers) need to perform complex ad-hoc analysis to understand usage trends and to determine how to improve the product. This requires low volume of queries (at most tens per second) and moderate latency (seconds to minutes) of complex queries (multiple levels of aggregations, set operations, analytic functions, joins, unpredictable patterns, manipulating nested / repeated data, etc.) over enormous volumes (trillions of rows) of data. Query patterns are highly unpredictable, although some standard data modeling techniques, such as star and snowflake schemas, can be used.

Historically, YouTube (and other similar products at Google and elsewhere) have relied on different storage and query backends for each of these needs: Dremel [33] for ad-hoc

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352121

analysis and internal dashboards, Mesa [27] and Bigtable [11] for external-facing high volume dashboards, Monarch [32] for monitoring site health, and Vitess [25] for embedded statistics in pages. But with exponential data growth and product enhancements came the need for more features, better performance, larger scale and improved efficiency, making meeting these needs increasingly challenging with existing infrastructures. Some specific problems we were facing were:

- Data needed to be loaded into multiple systems using different Extract, Transform, and Load (ETL) processes, leading to significant additional resource consumption, data quality issues, data inconsistencies, slower loading times, high development and maintenance cost, and slower time-to-market.
- Since each internal system uses different languages and API's, migrating data across these systems, to enable the utilization of existing tools, resulted in reduced usability and high learning costs. In particular, since many of these systems do not support full SQL, some applications could not be built by using some backends, leading to data duplication and accessibility issues across the organization.
- Several of the underlying components had performance, scalability and efficiency issues when dealing with data at YouTube scale.

To solve these problem, we built Procella, a new distributed query engine. Procella implements a superset of features required for the diverse workloads described above:

- **Rich API:** Procella supports an almost complete implementation of standard SQL, including complex multi-stage joins, analytic functions and set operations, with several useful extensions such as approximate aggregations, handling complex nested and repeated schemas, user defined functions, and more.
- **High Scalability:** Procella separates compute (running on Borg [42]) and storage (on Colossus [24]), enabling high scalability (thousands of servers, hundreds of petabytes of data) in a cost efficient way.
- **High Performance:** Procella uses state-of-the-art query execution techniques to enable efficient execution of high volume (millions of QPS) of queries with very low latency (milliseconds).
- **Data Freshness:** Procella supports high volume, low latency data ingestion in both batch and streaming modes, the ability to work directly on existing data, and native support for lambda architecture [35].

2. ARCHITECTURE

2.1 Google Infrastructure

Procella is designed to run on Google infrastructure. Google has one of world's most advanced distributed systems infrastructure, with several notable features, which have had a large impact on Procella's design:

- **Disaggregated storage:** All durable data must be stored in Colossus. No local storage is available to the system. Colossus, while being almost infinitely scalable, differs from local storage in a few important ways:

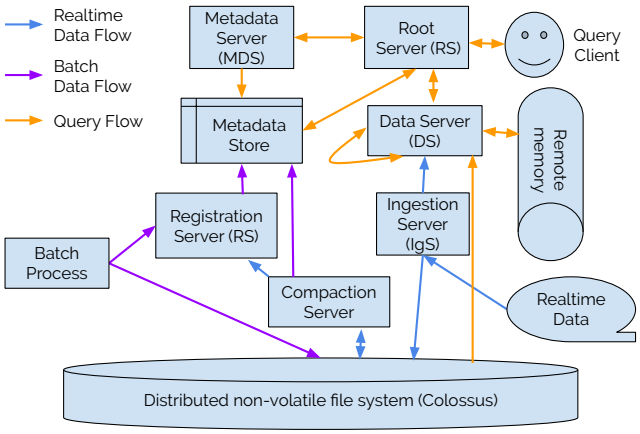

Figure 1: Procella System Architecture.

  - Data is immutable. A file can be opened and appended to, but cannot be modified. Further, once finalized, the file cannot be altered at all.
  - Common metadata operations such as listing and opening files can have significantly higher latency (especially at the tail) than local file systems, because they involve one or more RPCs to the Colossus metadata servers.
  - All durable data is remote; thus, any read or write operation can only be performed via RPC. As with metadata, this results in higher cost and latency when performing many small operations, especially at the tail.
- **Shared compute:** All servers must run on Borg, our distributed job scheduling and container infrastructure. This has several important implications:
  - Vertical scaling is challenging. Each server runs many tasks, each potentially with different resource needs. To improve overall fleet utilization, it is better to run many small tasks than a small number of large tasks.
  - Borg master can often bring down machines for maintenance, upgrades, etc. Well behaved tasks thus must learn to recover quickly from being evicted from one machine and brought up in a different machine. This, along with the absence of local storage, makes keeping large local state impractical, adding to the reasons for having a large number of smaller tasks.
  - A typical Borg cluster will have thousands of inexpensive machines, often of varying hardware configuration, each with a potentially different mix of tasks with imperfect isolation. Performance of a task, thus, can be unpredictable. This, combined with the factors above, means that any distributed system running on Borg must deploy sophisticated strategies for handling badly behaving tasks, random task failures, and periodic unavailability due to evictions.

Figure 1 illustrates the overall Procella architecture. Our architecture is structured from multiple components, each

executes in a distributed manner. When a component does not have any executing instance, the functionality it provides is unavailable (for example, even when data servers are down, ingestion and registration can function).

2.2 Procella's Data

2.2.1 Data Storage

Like most databases, Procella data is logically organized into tables. Each table's data is stored across multiple files (also referred to as *tablets* or *partitions*). Procella uses its own columnar format, Artus (Section 3.2) for most data, but also supports querying data stored in other formats such as Capacitor [34]. All durable data is stored in Colossus, allowing Procella to decouple storage from compute; i.e, the servers processing the data can be scaled to handle higher traffic independently of the size of the underlying data. Additionally, the same data can be served by multiple instances of Procella.

2.2.2 Metadata Storage

Like many modern analytical engines [18, 20], Procella does not use the conventional BTree style secondary indexes, opting instead for light weight secondary structures such as zone maps, bitmaps, bloom filters, partition and sort keys [1]. The metadata server serves this information during query planning time. These secondary structures are collected partly from the file headers during file registration, by the registration server, and partly lazily at query evaluation time by the data server. Schemas, table to file mapping, stats, zone maps and other metadata are mostly stored in the metadata store (in Bigtable [11] and Spanner [12]).

2.2.3 Table management

Table management is through standard DDL commands (CREATE, ALTER, DROP, etc.) sent to the registration server (RgS), which stores them in the metadata store. The user can specify column names, data types, partitioning and sorting information, constraints, data ingestion method (batch or realtime) and other options to ensure optimal table layout. For real-time tables, the user can also specify how to age-out, down-sample or compact the data; this is important for monitoring applications and for supporting a lambda architecture where batch data replaces real time data at a regular cadence.

Once a table object is created in the system, the table data can be populated, or ingested, in batch mode or realtime mode. Both of the ingestion modes differ in the optimizations they use.

2.2.4 Batch ingestion

Users generate data using offline batch processes (such as a MapReduce [15]) and register the data by making a DDL RPC to the RgS. This is the most common method employed by automated pipelines that refresh data at a regular cadence, such as hourly or daily. During the data registration step, the RgS extracts mapping of table to files and secondary structures (described in section 2.2.2 above) from file headers. We avoid scanning data during the registration step to enable fast data registration. However, we may utilize Procella data servers to lazily generate expensive secondary structures if the required indexing information is not present in the file headers, for example, if bloom filters are missing in externally generated data. There are no prerequisites on data organization for registering data with Procella [40]. In practice, however, for best performance, data should be laid out optimally using partitioning, sorting, etc.

The RgS is also responsible for sanity checks during the table and data registration step. It validates backwards compatibility of schemas, prunes and compacts complex schemas, ensures that file schemas are compatible with the table schemas registered by the user, etc.

2.2.5 Realtime ingestion

The ingestion server (IgS) is the entry point for real-time data into the system. Users can stream data into it using a supported streaming mechanism such as RPC or PubSub. The IgS receives the data, optionally transforms it to align with the table structure and appends it to a write-ahead log on Colossus. In parallel, it also sends the data to the data servers according to the data partitioning scheme for the table. The data is (temporarily) stored in the memory buffer of the data server for query processing. The buffers are also regularly checkpointed to Colossus to help in recovery if and after crash or eviction occurs, but this is best-effort and does not block query access to the data. The IgS (optionally) sends the data to multiple data servers for redundancy. Queries access all copies of the buffered data and use the most complete set.

The write-ahead log is compacted in the background by compaction tasks that provide durable ingestion.

Having the data follow two parallel paths allows it to be available to queries in a dirty-read fashion in seconds or even sub-second, while being eventually consistent with slower durable ingestion. Queries combine data from the in-memory buffers and the on-disk tablets, taking from the buffers only data that is not yet durably processed. Serving from the buffers can be turned off to ensure consistency, at the cost of additional data latency.

2.2.6 Compaction

The compaction server periodically compacts and re-partitions the logs written by the IgS into larger partitioned columnar bundles for efficient serving by the data servers. During the compaction process, the compaction server can also apply user defined SQL based logic (specified during table registration) to reduce the size of the data by filtering, aggregation, aging out old data, keeping only the latest value, etc. Procella allows a fairly rich SQL based logic to be applied during the compaction cycle, thus giving more control to the user to manage their real-time data.

The compaction server updates the metadata store (through the RgS) after each cycle, removing metadata about the old files and inserting metadata about the newly generated files. The old files are subsequently deleted by a background process.

2.3 Query Lifecycle

Clients connect to the **Root Server** (RS) to issue SQL queries. The RS performs query rewrites, parsing, planning and optimizations to generate the execution plan. To this end, it uses metadata such as schema, partitioning and index information from the **Metadata Server** (MDS) to prune the files to be read, as detailed in Section 3.4. It then orchestrates the query execution as it goes through the different stages

enforcing timing/data dependencies and throttling. To address the needs of executing complex distributed query plans (timing/data dependencies, diverse join strategies, etc.), the RS builds a tree composed of query blocks as nodes and data streams as edges (Aggregate, Execute Remotely, Stagger Execution, etc.) and executes it accordingly. This enables functionality such as shuffle (that requires timing dependency), uncorrelated subquery, subquery broadcast joins (data dependency) and multi-level aggregation. This graph structure allows several optimizations by inserting custom operations into the tree based on query structure. Once the RS receives the final results, it sends the response back to the client, along with statistics, error and warning messages, and any other information requested by the client.

The **Data Server** (DS) receives plan fragments from the RS or another DS and does most of the heavy lifting, such as reading the required data (from local memory, remote and distributed files in Colossus, remote memory using RDMA, or another DS), executing the plan fragment and sending the results back to the requesting RS or DS. As in most distributed SQL engines, Procella aggressively pushes compute close to the data. The plan generator ensures that filters, project expressions, aggregations (including approximate aggregations such as TOP, UNIQUE, COUNT DISTINCT and QUANTILE), joins, etc. are pushed down to the DS to the extent possible. This allows the DS to use the encoding-native functions for these operators for optimal performance. Data servers use Stubby [41] to exchange data with other data servers and RDMA for shuffle (Procella reuses the BigQuery shuffle library [4]).

3. OPTIMIZATIONS

Procella employs several techniques to achieve high query performance for various query patterns (lookups, reporting, ad-hoc queries, and monitoring). This section documents several of these techniques.

3.1 Caching

Procella achieves high scalability and efficiency by segregating storage (in Colossus) from compute (on Borg). However, this imposes significant overheads for reading or even opening files, since multiple RPCs are involved for each. Procella employs multiple caches to mitigate this networking penalty:

- **Colossus metadata caching:** To avoid file open calls to the Colossus name server, the data servers cache the file handles. The file handles essentially store the mapping between the data blocks and the Colossus servers that contain the data. This eliminates one or more RPC roundtrips on file opens.
- **Header caching:** The header (or sometimes, the footer) of columnar files contain various column metadata such as start offset, column size, and minimum and maximum values. Procella data servers cache the headers in a separate LRU cache, thus avoiding more round trips to Colossus.
- **Data caching:** The DS caches columnar data in a separate cache. The Procella data format, Artus, is designed so that data has the same representation in memory and on disk, making cache population fairly cheap. In addition, the DS caches derived information such as output of expensive operations and bloom filters. Note that since Colossus files are essentially immutable once closed, cache consistency is simply a matter of ensuring that file names are not reused.
- **Metadata caching:** Procella scales metadata storage by using a distributed storage system (Bigtable or Spanner) to store, and a distributed metadata service to serve metadata. However, this means that, often, metadata operations such as fetching table to file name mappings, schemas, and constraints can become a bottleneck. To avoid this, the metadata servers cache this information in a local LRU cache.
- **Affinity scheduling:** Caches are more effective when each server caches a subset of the data. To ensure this, Procella implements affinity scheduling to the data servers and the metadata servers to ensure that operations on the same data / metadata go to the same server with high probability. This means that each server is only responsible for serving a small subset of the data / metadata, which significantly improves cache hit ratio, dramatically reducing the time spent fetching remote data. An important aspect of Procella's scheduling is that the affinity is loose, i.e. the request can go to a different server (for example, if the primary is down or slow). When this happens, the cache hit probability is lower, but since the data / metadata itself is stored in a reliable durable storage (Bigtable, Spanner, or Colossus), the request still completes successfully. This property is crucial for high availability in the serving path in the face of process evictions, overloaded machines, and other issues associated with running in a large shared cluster.

The caching schemes are designed such that when there is sufficient memory, Procella essentially becomes a fully in-memory database. In practice, for our reporting instance, only about 2% of the data can fit in memory, but access patterns and cache affinity ensures that we get 99%+ file handle cache hit rate and 90% data cache hit rate.

3.2 Data format

The first implementation of Procella used Capacitor as the data format, which was primarily designed for large scans typical in ad-hoc analysis workloads. Since Procella aims to cover several other use cases requiring fast lookups and range scans (such as serving embedded statistics on high traffic pages), we built a new columnar file format called Artus, which is designed for high performance on both lookups and scans. Specifically, Artus:

- Uses custom encodings, avoiding generic compression algorithms like LZW. This ensures that it can seek to single rows efficiently without needing to decompress blocks of data, making it more suitable for small point lookups and range scans.
- Does multi-pass adaptive encoding; i.e. it does a first pass over the data to collect lightweight information (e.g. number of distinct values, minimum and maximum, sort order, etc.) and uses this information to determine the optimal encoding to use for the data. Artus uses a variety of methods to encode data: dictionary encoding with a wide variety of dictionary and

indexer types, run-length, delta, etc. to achieve compression within 2x of strong general string compression (e.g. ZSTD) while still being able to directly operate on the data. Each encoding has estimation methods for how small and fast it will be on the data supplied. Once the user has specified their objective function (relative value of size versus speed), Artus will use these methods to automatically choose the optimal encodings for the provided data.

- Chooses encodings that allow binary search for sorted columns, allowing fast lookups in $O(logN)$ time. Columns also support $O(1)$ seeks to a chosen row number, which means that a row with K columns matching a given primary key can be found in $O(logN + K)$. For columns that allow direct indexing, such as packing integers into a fixed number of bits, providing these $O(1)$ seeks is trivial, whereas for other columns, such as run-length-encoded columns, we must maintain additional 'skip block' information that records the internal state of the column every B rows, allowing us to 'skip' into the encoded data as if we had scanned through it until that point. Note that technically the seeks on these types of columns are $O(B)$ rather than $O(1)$ as we must iterate forward from the last skip block. The values used for B are generally quite small, such as 32 or 128, and represent a tradeoff between compression size and lookup speed. Fast row lookups are critical for lookup queries and distributed lookup joins, where we treat the right side table as a distributed hash map.

- Uses a novel representation for nested and repeated data types that is different from the method originally implemented in ColumnIO [33] and subsequently adopted by Capacitor, Parquet and others. We visualize a table's schema as a tree of fields. We store a separate column on disk for each of these fields (unlike rep/def, which only stores leaf fields). When ingesting a record, each time a parent field exists, we note the number of times each of its children occur. For optional fields, this is either 0 or 1. For repeated fields, this number is non-negative. Of particular interest here is that we do not record any information about fields whose parent does not exist; we are thus able to store sparse data more efficiently than rep/def, which always records data in each of its leaf fields. When reconstructing records, we essentially reverse the ingestion process, using the occurence information to tell us how many elements we need to copy. An important implementation detail is that by storing this occurence information in a cumulative manner separately from the actual values in the children we are able to still have $O(1)$ seeks even on nested and repeated data.

- Directly exposes dictionary indices, Run Length Encoding (RLE) [2] information, and other encoding information to the evaluation engine. Artus also implements various common filtering operations natively inside its API. This allows us to aggressively push such computations down to the data format, resulting in large performance gains in many common cases.

- Records rich metadata in the file and column header. Apart from the schema of the data, we also encode sorting, minimum and maximum values, detailed encoding information, bloom filters, etc. making many common pruning operations possible without the need to read the actual data in the column. In practice we often range-partition our files on a primary key, allowing us to use the metadata to prune file access based on the query filters.

- Supports storing inverted indexes. Inverted indices are commonly used at Google and elsewhere to speed up keyword search queries [7,38]. While we could use them for that purpose here, the main usage in Procella so far has been in experiment analysis. In experiment analysis, we have a small array of low-cardinality integers in each record, and we want to know which records contain a particular integer within their array, i.e. 'WHERE 123 IN ArrayOfExperiments', and then do further analysis on them. Evaluating this filter is normally the dominant cost in a query, because each array contains hundreds of elements. We avoid this cost by precomputing the inverted indices (the row offsets of records whose arrays contain each value), storing them on disk alongside regular columns, and then simply loading the appropriate indices based on the query. By storing these indices as Roaring bitmaps [10] we are able to easily evaluate typical boolean filters (i.e. 'WHERE 123 in ArrayOfExperiments OR 456 in ArrayOfExperiments') efficiently, without having to go through the normal evaluation pathway. In current production use cases we have found that experiment analysis queries have end to end latencies reduced by $\sim$ 500x orders of magnitude when we apply this technique.

Table 1: Queries for Artus benchmark

Query Id	*Query*
Q1	SUM(views) where pk = X: pure lookup, 1 matching row
Q2	SUM(views) where fk = X: moderate filter, 5000 matching rows
Q3	SUM(views): full scan, all 250k matching
Q4	SUM(views), fk GROUP BY fk: full scan with group by

Table 2: Artus Data Size

Format	*Capacitor*	$Artus_{Disk}$	$Artus_{memory}$
Size (KB)	2406	2060	2754

Figure 2 presents results of a simple benchmark comparing Artus with Capacitor on typical query patterns on a typical YouTube Analytics dataset. The numbers represent the best value observed in 5 runs of the test framework. The queries were run on a single file with ˜250k rows sorted on the *video id* and fully loaded into memory. Table 1 lists the queries used for the benchmark.

Table 2 shows Artus uncompressed in-memory size, as well as its LZW-compressed on-disk size, compared to Capacitor, for the data set used for the benchmark.

3.3 Evaluation Engine

High performance evaluation is critical for low latency queries. Many modern analytical systems today achieve this by using LLVM to compile the execution plan to native

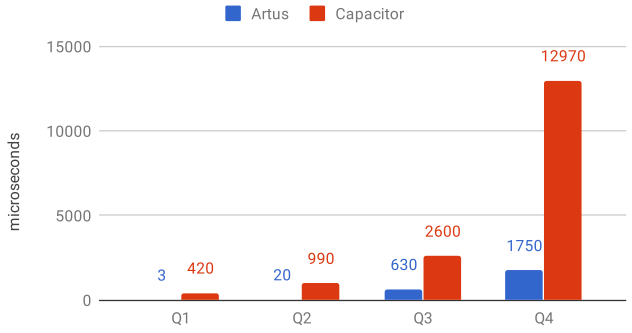

Figure 2: Artus vs Capacitor Performance.

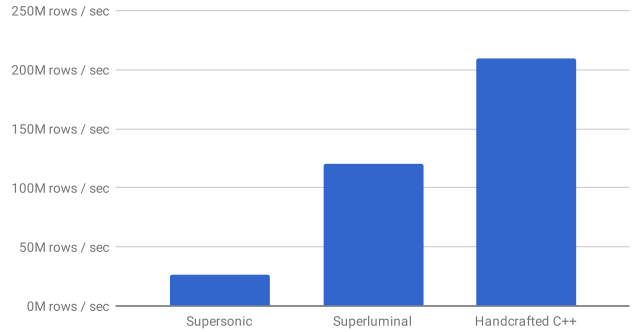

Figure 3: Superluminal vs Supersonic.

code at query time. However, Procella needs to serve both analytical and high QPS serving use cases, and for the latter, the compilation time can often become the bottleneck. The Procella evaluation engine, Superluminal, takes a different approach. Specifically, Superluminal:

- Makes extensive use of C++ template metaprogramming for compile time code generation. This allows utilizing many optimization techniques, such as partial application, to be applied to a single function implementation automatically, while avoiding large virtual call overheads.
- Processes data in blocks to take advantage of vectorized computation [37] and cache-aware algorithms. Block sizes are estimated to fit in the L1 cache memory for best cache friendliness. The code is carefully written to enable the compiler to auto-vectorize the execution [8].
- Operates on the underlying data encodings natively, and preserves them during function application wherever possible. The encodings are either sourced from the file format or generated during execution as needed.
- Processes structured data in a fully columnar fashion. No intermediatte representations are materialized.
- Dynamically combines filters and pushes them down the execution plan, all the way to the scan node, allowing the system to only scan the exact rows that are required for each column independently.

We benchmarked the aggregation performance of Superluminal against our first genertion row based evaluator and

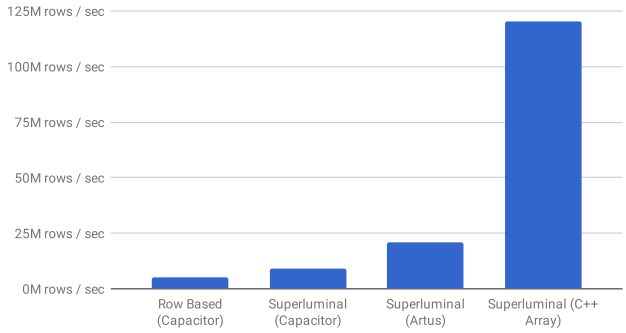

Figure 4: Superluminal performance on Artus & Capacitor.

the open source **Supersonic** engine using TPC-H query #1. The filter clause was removed to fairly compare only the aggregation performance, and the data was loaded into in-memory arrays to eliminate file format bias. The results (per CPU core) are in Figure 3. We also compared the performance on different formats, notably Capacitor, Artus and raw C++ arrays. The results (per CPU core) are in Figure 4. As evidenced from the numbers:

- Superluminal significantly improves on the legacy row based engine (almost twice as fast on Capacitor, and almost five times as fast on Artus, as shown in Figure 4). However, operating on raw C++ arrays is still significantly faster, albeit at the cost of much higher memory usage [1].
- On C++ arrays, Superluminal is almost 5 times faster than Supersonic, as shown in Figure 3; however, a very carefully hand-optimized native C++ implementation specific to the problem is about 50% faster. In our experience, a naive C++ implementation backed by a standard library hash map is an order of magnitude slower than Superluminal.

3.4 Partitioning & Indexing

Procella supports multi-level partitioning and clustering. Typically, most fact tables are date partitioned and within each date, clustered by multiple dimensions. Dimension tables would typically be partitioned and sorted by the dimension key. This enables Procella to quickly prune tablets that do not need to be scanned, and to perform co-partitioned joins, avoiding large shuffles. This is a critical feature, especially for our external reporting instance which needs to answer thousands of queries over tens of petabytes of data with milliseconds latency.

The MDS is responsible for efficient storage and retrieval of this information. For high scalability, MDS is implemented as a distributed service, with affinity scheduling for cache efficiency, as described in Section 3.1. In-memory structures used for caching the metadata are transformed from their storage format in Bigtable by using prefix, delta, RLE, and other encodings. This ensures that Procella can handle a very large volume of metadata (thousands of tables, billions of files, trillions of rows) in a memory efficient way.

[1] Artus files were about an order of magnitude smaller than the raw data for the TPC-H dataset.

Table 3: MDS Performance

Percentile	Latency (ms)	Tablets Pruned	Tablets Scanned
50	6	103745	57
90	11	863975	237
95	15	1009479	307
99	109	32799020	3974
99.9	336	37913412	7185
99.99	5030	38048271	11125

Table 3 illustrates the efficacy of the MDS tablets/partition pruning for our external reporting instance. The pruning itself is performed based on the filtering criteria within the query, e.g. ids based on hashing, date ranges, etc.

Once the MDS has effectively pruned the tablets from the plan, the individual leaf requests are sent to the data server. The data server uses bloom filters, min/max values, inverted indices, and other file level metadata to minimize disk access based on the query filters. This information is cached lazily in an LRU cache in each data server. This allows us to entirely skip evaluation for almost half of the data server requests in our main reporting instance.

3.5 Distributed operations

3.5.1 *Distributed Joins*

Joins are often a challenge for distributed query execution engines [6, 9, 21]. Procella has several join strategies that can be controlled explicitly using hints or implicitly by the optimizer based on data layout and size:

- **Broadcast:** This is the simplest join where one side is small enough so it can be loaded into the memory of each DS running the query. Small dimensions such as Country and Date often use this strategy.
- **Co-partitioned:** Often, the fact and dimension tables can be partitioned on the same (join) key. In these cases, even though the overall size of the RHS is large, each DS only needs to load a small subset of the data in order to do the join. This technique is commonly used for the largest dimension of a star schema (e.g. the Video dimension table).
- **Shuffle:** When both sides are large and neither is partitioned on the join key, data is shuffled on the join key to a set of intermediate servers.
- **Pipelined:** When the RHS is a complex query, but likey to result in a small set, the RHS is executed first and the result is inlined and sent to the LHS shards, resulting in a broadcast-like join.
- **Remote lookup:** In many situations, the dimension table (build side) may be large but partitioned on the join key, however the fact table (probe side) is not. In such cases, the DS sends remote RPCs to the DS serving the build side tablets to get the required keys and values for the joins. The RPC cost can often be high, so care must be taken to apply all possible filters and batch the keys into as few RPCs as possible. We also push down the projections and filters to the build side RPC requests to ensure that only the minimum required data is fetched. Procella can execute such lookup joins with high performance and efficiency because the underlying columnar format, Artus, supports fast lookups, without the need to first transform the data into in-memory hash table or similar structure.

3.5.2 *Addressing Tail Latency*

In modern distributed scale out systems which operate on cheap shared hardware, individual machines can often behave unreliably. This makes achieving low tail latency difficult. Various techniques exist to address this [14]. Procella employs similar techniques to achieve low tail latencies for its queries, but adapts them to its architecture. For example:

- Since all Procella data is present in Colossus, any file is accessible by any data server. The RS employs an effective backup strategy to exploit this. Specifically, it maintains quantiles of DS response latency dynamically while executing a query, and if a request takes significantly longer than the median, sends a backup RPC to a secondary data server.
- The RS limits the rates of, and batches, requests to the DS for large queries to avoid overwhelming the same data server with too many requests.
- The RS attaches a priority to each request it sends to the DS – typically, this is set to high for smaller queries and low for larger queries. The DS in turn, maintains separate threads for high and low priority requests. This ensures faster response for the smaller queries, so that one very large query cannot slow down the entire system.

3.5.3 *Intermediate Merging*

For queries that perform heavy aggregations, the final aggregation often becomes the bottleneck as it needs to process large amounts of data in a single node. To avoid this, we add an intermediate operator at the input of the final aggregation which buffers the data and dynamically spins up additional threads to perform intermediate aggregations, if the final aggregator cannot keep up with the responses from the leaf data servers. This significantly improves the performance of such queries, as can be seen in figure 5.

3.6 Query Optimization

3.6.1 *Virtual Tables*

A common technique used to maximize performance of low latency high QPS queries (needed for high volume reporting) is materialized views [3]. The core idea is to generate multiple aggregates of the underlying base table and choose the right aggregate at query time (either manually or automatically). Procella uses virtual tables to this end, which employs a similar approach to this problem with some additional features to ensure optimal query performance. Specifically, the Procella virtual table implementation supports:

- **Index-aware aggregate selection:** choose the right table(s) based not just on size but also on data organization such as clustering, partitioning, etc. by matching the filter predicates in the query to the table layout, thus, minimizing data scans.

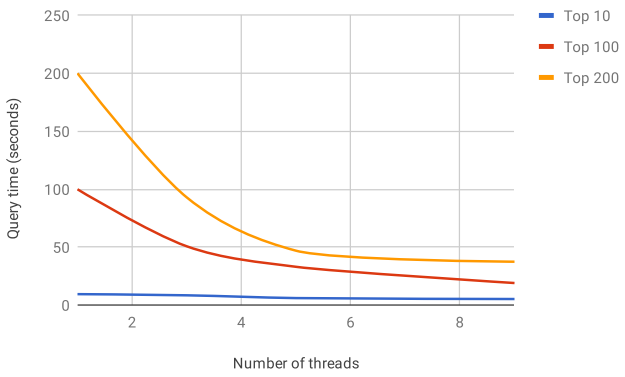

Figure 5: Intermediate merging performance for different $TOP(K)$ settings with increasing number of intermediate aggregation threads.

- **Stitched queries:** stitch together multiple tables to extract different metrics from each using `UNION ALL`, if they all have the dimensions in the query.
- **Lambda architecture awareness:** stitch together multiple tables with different time range availabilities using `UNION ALL`. This is useful for two purposes. First, for stitching together batch and realtime data where batch data is more accurate and complete, but arrives later than realtime data. Second, for making batch data consistently available at a specific cutoff point sooner by differentiating between batch tables that must be available to cover all dimension combinations vs. ones that are just useful for further optimization.
- **Join awareness:** the virtual table layer understands star joins and can automatically insert joins in the generated query when a dimensional attribute is chosen that is not de-normalized into the fact table.

3.6.2 *Query Optimizer*

Procella has a query optimizer that makes use of static and adaptive query optimization techniques [16]. At query compile time we use a rule based optimizer that applies standard logical rewrites (always beneficial) such as filter push down, subquery decorrelation, constant folding, etc. At query execution time we use adaptive techniques to select/tune physical operators based on statistics (cardinality, distinct count, quantiles) collected on a sample of the actual data used in the query.

Adaptive techniques have enabled powerful optimizations hard to achieve with traditional cost-based optimizers, while greatly simplifying our system, as we do not have to collect and maintain statistics on the data (especially hard when ingesting data at a very high rate) and we do not have to build complex estimation models that will likely be useful only for a limited subset of queries [19, 31]. We instead get statistics from the actual data as it flows through the query plan, our stats are thus equally accurate throughout the plan, in traditional optimizers estimates are more accurate at the leaves of the plan than closer to the root.

Our adaptive optimizer adds stats "collection sites" to the query plan and decides how to transform the un-executed portion of the plan based on this information. Currently we insert "collection sites" to all shuffle operations. A shuffle operation can be explained as a map-reduce. During the map phase we collect all rows that share a key into the same shard. During the reduce phase we perform computation on the shard. For example, for a group by, we map rows with the same group key to the same shard and then reduce the shard to compute the aggregates; for a join, we map the rows from the left and right tables with matching keys to the same shard and then reduce the shard to produce the joined rows. A shuffle defines a natural materialization point that facilitates the collection of stats. The stats are then used to decide how to partition the data including the number of reducers and the partition function. Stats are also used to rewrite the plan for the un-executed portion of the plan.

Adaptive Aggregation In this case we have two query fragments; a leaf fragment that computes partial aggregation on each partition of the data, and a root fragment that aggregates the partial results. We estimate the number of records that the first fragment will emit by computing the partial aggregate on a subset of the partitions and deciding the number of shards to satisfy a target number of rows per shard (e.g. 1 million).

Adaptive Join. When we encounter a join we estimate the keys that will participate in the join from the LHS and RHS. We use a data structure that summarizes the keys for each partition. The data structure holds the count of keys, the min and max values per partition, and a bloom filter if the number of keys is less than a few million. In most cases this data structure is collected from the entire input, not just from a sample. We support the following join optimizations:

- **Broadcast**. When one of the sides is small (*build-side*) (e.g. less than 100,000 records) we broadcast it (using RDMA) to every partition of the *probe-side*. In this case the plan is rewritten to avoid the shuffle.
- **Pruning**. When the keys on one of the sides (*filter-side*) can be summarized with a bloom filter of size $O(10MB)$ with a false positive rate of about 10%; we use this bloom filter to prune the other side (*prune-side*). The map operation only writes records from the *prune-side* that pass the filter, we only map records with a high probability of joining (based on the bloom filter false positive ratio).
- **Pre-shuffled**. If one side of the join (*partitioned-side*) is already partitioned on the join key, we use the map operation to partition only the other side *map-side* to match the *partition-side* partition. For example, if the *partition-side* has partitions [min,max] = [1,5] [10,50] [150,200] we map records from the *map-side* to match these intervals (we may overpartition if the *map-side* is large). The *partition-side* could initially have hundreds of partitions, but after applying filters only these three ranges of values will actually participate in the join. This optimization is very frequently applicable in fact to dimension joins as dimension tables are partitioned on the join key. Note that if both left and right sides are partitioned on the join key, at compile time, we select a lookup join strategy (i.e. the shuffle is avoided).
- **Full Shuffle**. If no other join optimization applies, we map records from the left and right sides of the join. We automatically determine the number of shards based on the sum of records from the left and right sides.

Adaptive Sorting. When handling an order by statement, we first issue a query to estimate the number of rows that need to be sorted and use this information to determine a target number of shards n; We then issue a second query to estimate n quantiles that are used to range partition the data during the map operation. During the reduce operation we locally sort the data in each shard. Adaptive sorting allows us to reuse our shuffle infrastructure to order large datasets efficiently.

Limitations of Adaptive optimization. Adaptive optimization works well for large queries where we only add around 10% of overhead for stats collection. But for small queries (i.e. queries that execute in $O(10ms)$), the overhead of adaptive optimization can be large (around $2X$ penalty). For low latency queries we allow the user to specify query hints that fully define the execution strategy without the adaptive optimizer. We have found that for ad-hoc queries, users are willing to tolerate a slow down of fast queries (e.g. 10ms to 20ms) in exchange for better performance on large queries and overall much better worst-case behavior.

Another limitation of our current adaptive framework is that it is not yet used for join ordering. We are working on an extension to the framework that uses a standard dynamic programming algorithm that collects table level statistics with adaptive queries. This can be a good compromise between the number of the executed stats queries and the accuracy of the estimates.

3.7 Data Ingestion

For optimal query processing performance, it is important to create datasets that are suitably partitioned, clustered, and sorted. At the same time, integrating loading into the query engine reduces scalability and flexibility for teams who want to import large (often petabytes) of existing data quickly into Procella. To address this, Procella provides an offline data generation tool. The tool takes the input and output schemas and data mapping, and executes an offline MapReduce based pipeline to create data in Procella optimized format and layout. Since the tool can run on cheap low priority batch resources on any data center, it vastly improves the scalability of ingestion of large data sets into Procella, and ensures that expensive production resources are only used for serving queries.

The tools can apply several optimizations during data generation such as uniform and suitably sized partitioning, aggregation, sorting, placing data for hot tables in SSD, replicating, encrypting, etc. Note that the use of the tool is optional; Procella can accept data generated by any tool as long as it is in a supported format.

3.8 Serving Embedded Statistics

Procella powers various embedded statistical counters, such as views, likes, and subscriptions, on high traffic pages such as YouTube's watch and channel pages. The (logical) query is simply: `SELECT SUM(views) FROM Table WHERE video_id = N`, and the data volumes are relatively small: up to a few billion rows and a small number of columns per row. However, each Procella instance needs to be able to serve over a million QPS of such queries with millisecond latency. Further, the values are being rapidly updated and we need to be able to apply the updates in near real-time. Procella solves this problem by running these instances in "stats serving" mode, with specialized optimizations for such workloads:

- When new data is registered, the registration server notifies the data servers (both primary and secondary) that new data is available, so that it can be loaded in memory immediately, instead of being lazily loaded on first request. This ensures that there is no remote disk access in the serving path, even if a few leaf servers are down. This ends up using more RAM but since we are dealing with relatively small volumes of data, this is a reasonable trade-off.

- The MDS module is compiled into the Root Server (RS). This saves RPC communication overheads between these sub-systems, allowing the RS to be efficient at high QPS.

- all metadata are fully preloaded and kept up to date asynchronously to avoid having to remotely access metadata at query time and thus suffer from higher tail latencies. The system is set up to have slightly stale metadata still give the right answers. The stats instances have a relatively small number of tables so this overhead is justifiable.

- Query plans are aggressively cached to eliminate parsing and planning overhead. This is very effective since the stats query patterns are highly predictable.

- The RS batches all requests for the same key and sends them to a single pair of primary and secondary data servers. This minimizes the number of RPCs required to serve simple queries. This means that single key queries have exactly 2 outgoing RPCs where we use the fastest response to minimize tail latencies.

- The root and data server tasks are monitored for unusually high error rates and latencies compared to the other tasks of the same server in the same data center. The problematic outlier tasks are automatically moved to other machines. This is useful because of the imperfect isolation on Borg.

- Most of the expensive optimizations and operations such as adaptive joins and shuffles are disabled to avoid related overheads and production risks.

4. PERFORMANCE

Procella is designed for high performance and scalability on flexible workloads over shared infrastructure. In this section, we present benchmarks and production serving numbers for three most common workloads, namely, ad-hoc analysis, real-time reporting, and serving stats counters.

4.1 Ad-hoc analysis

For ad-hoc analysis, we chose to use the TPC-H queries. TPC-H is a widely used and published benchmark for database systems that represents a typical normalized database and analytical workload.

We ran the TPC-H 1T (SF: 1000) and 10T (SF: 10000) queries on an internal Procella instance with 3000 cores and 20 TB RAM, running on Borg. Data was generated to our file format (Artus), partitioned, and sorted, and stored in Colossus. Caches were warmed up prior to executing the

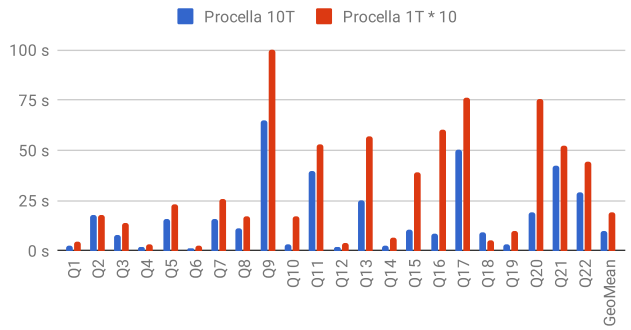

Figure 6: TPC-H Scalability - 1T Expected Execution Time vs. 10T Execution Times.

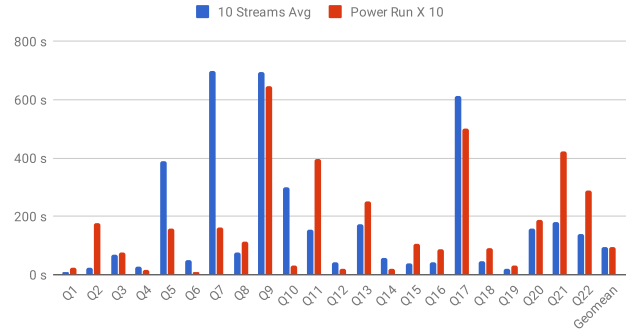

Figure 7: TPC-H 10T Per Query Throughput Execution times.

benchmarks. For 10T, we ran a single stream (power run) as well as ten parallel streams (throughput run) [2].

Figure 6 compares the execution time of individual TPC-H queries over 1T and 10T of data using the same instance. As shown, the 1T run had a geomean of about 2 seconds, while the power run on 10T had a geomean of about 10 seconds. The system, thus, scales well with data volume. The latency scaled sub-linearly, mainly because the 10T run was able to utilize more parallelism.

Figure 7 captures Procella performance for the TPC-H 10T throughput run with ten parallel streams. As shown, Procella scales well under load, i.e the performance degrades quite smoothly as the number of parallel queries increases. There was a fair amount of variance, however, with some queries (e.g. Q7) degrading much more than expected, while some queries (e.g. Q2) performing significantly better than expected. The main bottleneck we established was inter-node data transfer overhead (RPC, serialization & deserialization, etc.). This is somewhat expected of systems that scale horizontally by using many (hundreds or thousands of) relatively small tasks instead of a small number (ten or less) of large tasks.

4.2 Real-time reporting

Procella is used to serve many reporting and dashboarding solutions both internally and externally. Some of the external applications are YouTube Analytics, YouTube Music Insights, Firebase Performance Analytics and Play Developer Console.

In this section, we present results from the most popular application, which is youtube.com/analytics, known internally as YouTube Analytics (YTA). YTA runs more than a billion queries per day over tens of petabytes of data with tens of thousands of unique query patterns (dimensional breakdowns, filters, joins, aggregations, analytical functions, set operators, approximate functions, etc.). However, because each creator is restricted to their own data, each query is typically constrained to a small slice of the overall dataset. The data needs to be provided in real time; typically end to end delay is under a minute, but we also need to support large historical reports covering years of data. There is also an interface to do bulk extraction of data; the largest of such reports produce over a hundred gigabytes of output.

For redundancy, we maintain five instances of Procella for this use-case, with at least three fully serving. Each instance has about 6000 cores and 20 TB of RAM.

Table 4 provides a glimpse into the scale of the Procella instance that serves YTA (and a few other smaller applications). Table 5 captures the distribution of latency vs various aspects of the system.

As evidenced, Procella performs and scales very well when handling high volume SQL queries of simple to moderate complexity, on both real time and batch data, providing sub-second response for the 99%+ of the requests from creators, app developers and other external users of the system.

Table 4: YTA Instance Scale

Queries executed	1.5+ billion per day
Queries on realtime data	700+ million per day
Leaf plans executed	250+ billion per day
Rows scanned	80+ quadrillion per day
Rows returned	100+ billion per day
Peak scan rate per query	100+ billion rows per second
Schema size	200+ metrics and dimensions
Tables	300+

Table 5: YTA Instance Performance

Property	50%	99%	99.9%
E2E Latency	25 ms	412 ms	2859 ms
Rows Scanned	16.5M	265M	961M
MDS Latency	6 ms	119 ms	248 ms
DS Latency	0.7 ms	73 ms	220 ms
Query Size	300B	4.7KB	10KB

4.3 Embedded Statistics

The stats Procella instances power various embedded statistical counters, such as views, likes, and subscriptions, serving millions of QPS (hundreds of billions of queries per day) with millisecond latency from more than ten data centers around the world. Table 6 captures the latency profile in production of the statistics serving instances on a typical day.

As evidenced, in this configuration, Procella achieves performance comparable to NoSQL key value stores while supporting SQL based queries on the data.

[2]Note that these benchmarks are not fully compliant with the TPC-H rules, e.g. we hand tuned some queries for best performance in our environment. Thus, the results are not directly comparable to published TPC-H data but provide an insight on the raw query processing speed of the system.

Table 6: Statistics Serving Latency

Percentile	Latency
50%	1.6 ms
90%	2.3 ms
95%	2.6 ms
99%	3.3 ms
99.9%	14.8 ms
99.99%	21.7 ms

5. RELATED WORK

5.1 Google Technologies

- **Dremel** is an exa-scale columnar SQL query engine that is optimized for large complex ad-hoc queries. Dremel works on many backends such as ColumnIO, Capacitor, and Bigtable. The Dremel engine powers Google's BigQuery. Procella shares many properties with Dremel such as the RPC protocol, the SQL parser, low level storage (both use Colossus), compute (both use Borg), etc. However, there are significant design differences, for example, extensive use of stateful caching (Dremel is mostly stateless), separate instances where each is optimized for different use cases (Dremel is a global shared service across all users), use of indexable columnar formats optimized for lookups (Capacitor, unlike Artus, does not have indexes), etc. These make Procella suitable for many additional workloads (e.g high QPS reporting and lookup queries)

- **Mesa** is a peta-scale data storage and query system built at Google. Mesa is designed mainly for storing and querying aggregated stats tables, with a predominantly delta based low latency (measured in minutes) data ingestion model. Mesa does not itself support SQL; instead, F1 provides a flexible SQL interface on top of Mesa storage, using the lower level Mesa API.

- **F1** [39] includes a federated distributed query engine (similar to Dremel) that runs on top of many Google backends (Mesa, ColumnIO, Bigtable, Spanner, etc.) and provides support for scalable distributed SQL queries. F1 aims to support OLTP, ad-hoc, and reporting use-cases, but takes a fundamentally different approach from Procella, namely query federation. Specifically, F1 aims to decouple the execution engine from storage and instead exploits properties of the underlying storage subsystem optimally to serve diverse workloads. For example, OLTP workloads typically query F1 and Spanner databases. Low latency reporting applications use queries over F1 and Mesa. Analytical and ad hoc queries run over data in various sources, including Mesa, F1, and Capacitor files on Colossus. Procella, on the other hand, aims to serve diverse workloads (excluding OLTP) using the same storage and data format, by tightly coupling storage, data format, and execution into a single system.

- **PowerDrill** [28] is a columnar in-memory distributed SQL query engine optimized primarily for serving a relatively small QPS of large but relatively simple queries which can be trivially parallelized in a distributed tree execution architecture. The predominant use-case for PowerDrill is fast analysis of logs data and powering internal dashboards.

- **Spanner** is a distributed SQL database typically used by OLTP applications. Spanner shares many of the same underlying design elements (and hence similar performance characteristics) as Bigtable while providing better data consistency, atomic transactions, and a SQL based query interface. Spanner provides external consistency with realtime data and is often used as source-of-truth data store; however it does not aim to serve the ad-hoc or real-time reporting use-cases.

5.2 External Technologies

5.2.1 Ad-hoc analysis

Presto [22] is an open source query engine originally developed at Facebook that is similar in design to Dremel. Presto is now also available as a cloud service from Amazon as **Athena**. **Spark SQL** [5], an open source SQL engine that runs on the Spark framework, has also gained significant popularity, especially in organizations with large investment in Spark. **Snowflake** [13], a popular analytical database, segregates storage and compute, while adding several other features such as local caching and centralized metadata management, similar to Procella. **Redshift** [26] is a popular distributed database available from Amazon that tightly couples storage and compute on the same VMs, though it is possible to also access external data in **S3** through **Spectrum**. These systems excel at low QPS ad-hoc analysis and dashboarding, but do not offer high QPS low latency serving solutions, or high bandwidth real time streaming ingestion.

5.2.2 Real time reporting

Both **Druid** [43] and **Pinot** [17] share some of Procella's characteristics such as columnar storage, mixed streaming and batch ingestion modes, distributed query execution, support for tablet and column pruning, etc. that enable them to support low latency high QPS serving use cases. **ElasticSearch** has some structured query capabilities such as filtering and aggregation using a custom API and has been used in several places to power low latency real time dashboards. These engines have limited SQL support and are overall unsuitable for ad-hoc analysis. Companies like **Amplitude** and **Mixpanel** have built custom backends to satisfy the performance requirements of high volume low latency real time reporting. **Apache Kylin**, attempts to solve this problem differently by building aggregates from data cubes and storing them in a key value store such as HBase for higher performance for pre-defined query patterns that do not need large scans.

5.2.3 Monitoring

Stackdriver and **Cloudwatch** are two popular cloud monitoring services - they have their own custom backends and expose only a limited API to their users. Facebook has developed **Gorilla** [36], a real-time time-series monitoring database, now open-sourced as **Beringei** [23]. Beringei supports a custom time series API. **InfluxDB** [30] is a popular open source choice for time series real time analytics with flexible queries. **OpenTSDB** is an open source time series database popular for monitoring purposes that uses

a key value store such as **HBase** to store metrics keyed by time and user specified dimension keys and provides a custom API to extract time series data for charting and monitoring purposes. These systems do not offer efficient execution of complex queries over large data volumes, and are thus unsuitable for ad-hoc analysis.

5.2.4 Serving Statistics

Key value stores such as **HBase** are often used as backends for serving high QPS of updates and lookups simultaneously. They are often deployed with a caching layer. This method, however, typically does not provide flexible update modes such as delta mode for stats or HyperLogLog [29] for uniques, nor flexible SQL based query APIs that are required for analytical processing.

6. FUTURE WORK

Procella is a relatively young product, and there are a number of areas we are actively working on:

- As Procella gains wider adoption inside YouTube and Google, various usability aspects such as better tooling, better isolation and quota management for hybrid loads, better reliability, availability in more data centers, better documentation, better monitoring and such are gaining importance. The team is increasingly spending more effort to make it easier for users to adopt Procella.
- We continue to spend significant engineering effort on performance, efficiency, and scale – for example on data format (Artus), evaluation engine (Superluminal), structured operators and distributed operators such as joins. We are actively working on improving the optimizer, focused mainly on adaptive techniques to achieve optimal and predictable performance on large complex queries.
- We continue to work on enhancing SQL support, and on various extensions such as full text search, time-series, geo-spatial, and graph queries.

7. CONCLUSIONS

In this paper, we presented Procella, a SQL query engine that successfully addresses YouTube's need for a single product to serve a diverse set of use cases, such as real-time reporting, serving statistical counters, time-series monitoring, and ad-hoc analysis, all at very high scale and performance. We presented the key challenges of building such as system on Google's infrastructure, described the overall architecture of the system, and several optimizations we implemented to achieve these goals. Many Procella components (such as Artus and Superluminal) and design elements (such as affinity scheduling and layered caching) are being adopted by other Google products such as BigQuery, F1, and logs. Many of these techniques, we believe, are applicable to other similar distributed platforms such as AWS and Azure.

Procella has been successfully applied at Google to solve problems belonging to all four categories:

- **Reporting:** Virtually all of external reporting and internal dashboards at YouTube are powered by Procella (billions of queries per day). Procella is also powering similar external facing reporting applications for Google Play and Firebase, among others.
- **Embedded statistics:** Several high traffic stats counters on YouTube such as views, likes, and subscriptions are powered by Procella, totalling over 100 billion queries per day.
- **Ad-hoc analysis:** Procella is increasingly being adopted at YouTube for large complex ad-hoc analysis use cases over petabyte scale tables.
- **Monitoring:** Procella powers various real time anomaly detection, crash analysis, experiment analysis, and server health related dashboards at Google.

Procella thus enables YouTube to have a unified data stack that is highly scalable, performant, feature-rich, efficient, and usable for all types of data driven applications. Procella has now been in production for multiple years. Today, it is deployed in over a dozen data centers and serves hundreds of billions of queries per day over tens of petabytes of data, covering all four use cases, at YouTube and several other Google products.

8. ACKNOWLEDGEMENTS

We would like to thank the Dremel team, especially Mosha Pasumansky, for working closely with us on using the Capacitor format and enabling reuse of the Dremel interfaces. Additionally, We would like to thank the Google SQL team for developing and sharing the SQL parser and the test suite. We thank the YouTube management, especially Joel Truher, John Harding, Christian Kleinerman, Vaishnavi Shashikant, Eyal Manor, and Scott Silver for supporting the Procella project. We also thank Xing Quan for supporting the product requirements of Procella. Finally, we thank the many reviewers who helped review this paper, especially Andrew Fikes, Jeff Naughton and Jeff Shute.

9. REFERENCES

[1] D. Abadi, P. Boncz, and S. o. Harizopoulos. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[2] D. Abadi, Madden, et al. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682. ACM, 2006.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] H. Ahmadi. In-memory query execution in Google BigQuery, 2016.

[5] M. Armbrust, R. S. Xin, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[6] C. Barthels, I. Müller, et al. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 10(5):517–528, 2017.

[7] T. A. Bjørklund, J. Gehrke, and Øystein Torbjørnsen. A Confluence of Column Stores and Search Engines: Opportunities and Challenges, 2016.

[8] B. Bramas. Inastemp: A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization. *Scientific Programming*, 2017.

[9] N. Bruno, Y. Kwon, and M.-C. Wu. Advanced Join Strategies for Large-scale Distributed Computation. *PVLDB*, 7(13):1484–1495, 2014.

[10] S. Chambi, Lemire, et al. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[11] F. Chang, J. Dean, et al. Bigtable: A distributed storage system for structured data. *TOCS*, 26(2):4, 2008.

[12] J. C. Corbett, J. Dean, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3):8, 2013.

[13] B. Dageville, T. Cruanes, et al. The Snowflake Elastic Data Warehouse. In *SIGMOD*, SIGMOD '16, pages 215–226, New York, NY, USA, 2016. ACM.

[14] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Found. Trends databases*, 1(1):1–140, 2007.

[17] K. G. Dhaval Patel, Xaing Fu and P. N. Naga. Real-time Analytics at Massive Scale with Pinot, 2014.

[18] R. Ebenstein and G. Agrawal. Dsdquery dsi-querying scientific data repositories with structured operators. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 485–492. IEEE, 2015.

[19] R. Ebenstein and G. Agrawal. Distriplan: An optimized join execution framework for geo-distributed scientific data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 25. ACM, 2017.

[20] R. Ebenstein, G. Agrawal, J. Wang, J. Boley, and R. Kettimuthu. Fdq: Advance analytics over real scientific array datasets. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 453–463. IEEE, 2018.

[21] R. Ebenstein, N. Kamat, and A. Nandi. Fluxquery: An execution framework for highly interactive query workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1333–1345. ACM, 2016.

[22] Facebook Inc. Presto: Distributed SQL Query Engine for Big Data, 2015.

[23] Facebook Inc. Beringei: A high-performance time series storage engine, 2016.

[24] A. Fikes. Storage Architecture and Challenges, 2010.

[25] Google, Inc. Vitess: Database clustering system for horizontal scaling of MySQL, 2003.

[26] A. Gupta, D. Agarwal, et al. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, SIGMOD '15, pages 1917–1923, New York, NY, USA, 2015. ACM.

[27] A. Gupta, F. Yang, et al. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[28] A. Hall, O. Bachmann, et al. Processing a Trillion Cells per Mouse Click. *PVLDB*, 5:1436–1446, 2012.

[29] S. Heule, M. Nunkesser, and A. Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In *EDBT*, pages 683–692, Genoa, Italy, 2013.

[30] InfluxData Inc. InfluxDB: The Time Series Database in the TICK Stack, 2013.

[31] Lohman, Guy. Is query optimization a "solved" problem?, 2014.

[32] R. Lupi. Monarch, Google's Planet Scale Monitoring Infrastructure, 2016.

[33] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.

[34] Mosha Pasumansky. Inside Capacitor, BigQuery's next-generation columnar storage format, 2016.

[35] Nathan Marz. Lambda Architecture, 2013.

[36] T. Pelkonen, S. Franklin, et al. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[37] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[38] I. I. Prakash Das. Part 1: Add Spark to a Big Data Application with Text Search Capability, 2016.

[39] B. Samwel, J. Cieslewicz, et al. F1 Query: Declarative Querying at Scale. *PVLDB*, 11(12):1835–1848, 2018.

[40] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1239–1242. ACM, 2008.

[41] Varun Talwar. gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments, 2016.

[42] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, page 18, Bordeaux, France, 2015. ACM.

[43] F. Yang, E. Tschetter, et al. Druid: A Real-time Analytical Data Store. In *SIGMOD*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.