Dremel: A Decade of Interactive SQL Analysis at Web Scale[*]

Sergey Melnik, Andrey Gubarev,
Jing Jing Long, Geoffrey Romer,
Shiva Shivakumar, Matt Tolton,
Theo Vassilakis
Original authors of VLDB 2010 Dremel paper

Hossein Ahmadi, Dan Delorey,
Slava Min, Mosha Pasumansky,
Jeff Shute
Google LLC
dremel-tot-paper@google.com

ABSTRACT

Google's Dremel was one of the first systems that combined a set of architectural principles that have become a common practice in today's cloud-native analytics tools, including disaggregated storage and compute, in situ analysis, and columnar storage for semistructured data. In this paper, we discuss how these ideas evolved in the past decade and became the foundation for Google BigQuery.

PVLDB Reference Format:
Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *PVLDB*, 13(12): 3461-3472, 2020.
DOI: https://doi.org/10.14778/3415478.3415568

1. INTRODUCTION

Dremel is a distributed system for interactive data analysis that was first presented at VLDB 2010 [32]. That same year, Google launched BigQuery, a publicly available analytics service backed by Dremel. Today, BigQuery is a fully-managed, serverless data warehouse that enables scalable analytics over petabytes of data.[1] It is one of the fastest growing services on the Google Cloud Platform.

A major contribution of papers originating from the industry in the past decade, including the Dremel paper, is to demonstrate what kind of systems can be built using state-of-the-art private clouds. This body of work both reduced the risk of exploring similar routes and identified viable directions for future research. Introducing the journal version of the paper [33], Mike Franklin pointed out that it was "eye-opening" to learn that Google engineers routinely analysed massive data sets with processing throughputs in the range of 100 billion records per second [20]. His main take-away was that simply throwing hardware at the problem was not sufficient. Rather, it was critical to deeply understand the structure of the data and how it would be used. Franklin made an on-the-mark prediction that the data volumes described in the paper would soon become relevant to more organizations, given how quickly the "bleeding edge" becomes commonplace in our field. He also called out various opportunities for optimizations and improvements.

This paper focuses on Dremel's key ideas and architectural principles. Much of the overall system design stood the test of time; some of these principles turned into major industry trends and are now considered best practices. Stated in terms of the technology trends highlighted in the recently published Seattle Report on Database Research [1], the main ideas we highlight in this paper are:

- *SQL*: [1] reports that all data platforms have embraced SQL-style APIs as the predominant way to query and retrieve data. Dremel's initial SQL-style dialect got generalized as ANSI-compliant SQL backed by an open-source library and shared with other Google products, notably Cloud Spanner.[2]

- *Disaggregated compute and storage*: The industry has converged on an architecture that uses elastic compute services to analyze data in cloud storage. This architecture decouples compute from storage, so each can scale independently.

- *In situ analysis*: Data lake repositories have become popular, in which a variety of compute engines can operate on the data, to curate it or execute complex SQL queries, and store the results back in the data lake or send results to other operational systems. Dremel's use of a distributed file system and shared data access utilities allowed MapReduce and other data processing systems at Google to interoperate seamlessly with SQL-based analysis.

- *Serverless computing*: As an alternative to provisioned resources, the industry now offers on-demand resources that provide extreme elasticity. Dremel was built as a fully-managed internal service with no upfront provisioning and pay-per-use economics. This concept was successfully ported to BigQuery.

- *Columnar storage*: While use of columnar storage in commercial data analytic platforms predates the Dremel paper, Dremel introduced a novel encoding for nested data that generalized the applicability of column stores to nested relational and semistructured data.

This paper is structured around the above trends. For each, we explain the original motivation and examine the evolution of

[*] Invited contribution for the VLDB 2020 Test of Time Award given to the VLDB 2010 paper "Dremel: Interactive Analysis of Web-Scale Datasets" [32]

[1] https://cloud.google.com/bigquery

[2] https://cloud.google.com/spanner

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415568

the idea within Google and in public clouds. Orthogonal to these trends, the Seattle Report emphasizes that providing interactive response times over Big Data remains an open challenge, since high latency reduces the rate at which users make observations, draw generalizations, and generate hypotheses. We discuss how Dremel tackled latency while pushing forward the trends listed above.

Several leading engineers from today's BigQuery team were invited to co-author this retrospective paper, allowing us to shed more light onto the technical advances made in the intervening years.

2. EMBRACING SQL

Google was an early pioneer of the Big Data era. In the early 2000s, the company developed a new ethos around distributed infrastructure built on massive fleets of cheap, unreliable, commodity servers. GFS [21] and MapReduce [19] became the standard ways to store and process huge datasets. MapReduce made it easy to process data in parallel on thousands of machines, hiding most concerns about communications, coordination, and reliability. A custom language, Sawzall [36], was developed to make MapReduces somewhat easier to write than using C++ directly. NoSQL storage systems such as BigTable [13] also became the default for managing transactional data at scale.

The conventional wisdom at Google was "SQL doesn't scale", and with a few exceptions, Google had moved away from SQL almost completely. In solving for scalability, we had given up ease of use and ability to iterate quickly.

Dremel was one of the first systems to reintroduce SQL for Big Data analysis. Dremel made it possible for the first time to write a simple SQL query to analyze web-scale datasets. Analysis jobs that took hours to write, build, debug, and execute could now be written in minutes and executed in seconds, end-to-end, allowing users to interactively write, refine and iterate on queries. This was a paradigm shift for data analysis. The ability to interactively and declaratively analyze huge datasets, ad hoc, in dashboards, and in other tools, unlocked the insights buried inside huge datasets, which was a key enabler for many successful products over the next decade.

Dremel's SQL dialect was quirky but included some critical innovations—notably, first-class support for structured data. Protocol Buffers [37] were used pervasively at Google. Nearly all data passed between applications or stored on disk was in Protocol Buffers. Typical log records encoded details across thousands of nested and repeated fields. Dremel made it easy to query that hierarchical data with SQL.

Hierarchical schemas were a big departure from typical SQL schema design. Textbook normal forms would use many tables, and query-time joins. Avoiding joins was a key enabler for Dremel's scalable and fast execution. (Dremel initially had no join support, and was successful for years with only limited join support.) Denormalizing related data into one nested record was common in Google's datasets; hierarchical schemas made it unnecessary to flatten or duplicate any data, which would have increased storage and processing cost.

The F1 [41] project started in 2009, driving a parallel re-emergence of SQL in transactional Big Data systems at Google. The Ads team was tired of trying to scale core data in sharded MySQL, while moving larger datasets out to scalable systems such as Mesa [23] and BigTable. F1 was built as a hybrid of a traditional SQL database and a massively distributed storage system like BigTable. By 2013, Ads had moved completely to F1, and other OLTP-focused applications followed, also seeing the advantage of returning from NoSQL to SQL. Most transactional database functionality from F1 was later adopted by Spanner, which now backs most transactional applications at Google. F1 continues to focus on new SQL query use cases and optimizations, including HTAP with F1 Lightning [46], and federating across dozens of other specialized storage systems.

SQL finally became pervasive at Google, across widely used systems such as Dremel, F1, and Spanner, and other niche systems such as PowerDrill [24], Procella [15], and Tenzing [16]. Google was also beginning to enter the cloud business with an early version of BigQuery based on Dremel. All of these systems had their own SQL implementations and dialects. Users often utilized several of these systems and had to learn multiple idiosyncratic and non-standard dialects.

To address this complexity and improve on inevitable mistakes we made while designing SQL dialects ad hoc, we started the GoogleSQL [8] project, unifying on one new SQL implementation we could share across all SQL-like systems. This framework included:

- A new SQL dialect, complying with the ANSI standard with extensions for critical features such as querying structured data.
- A common parser, compiler front-end, and resolved algebra.
- A shared library of SQL functions.
- A simple reference implementation, demonstrating correct behavior.
- Shared test libraries, including compliance tests ensuring engine behavior matches our language specification.
- Other essential tools like a random query generator and a SQL formatter.

All SQL systems at Google, including BigQuery, Cloud Spanner, and Cloud DataFlow, have now adopted this common dialect and framework. Users benefit from having a single, standard-compliant and complete dialect they can use across many systems. These common SQL libraries are now available in open source as ZetaSQL.[3]

Sadly, while there is an ANSI standard for SQL, this standard is of limited use in practice. Since the standard is underspecified and lacks key functionality, every engine must make its own decisions on how to extend the standard, and which of several existing and mutually contradictory precedents from other engines to follow. For engines, this adds tremendous complexity and ambiguity when implementing SQL, with many tradeoffs to consider. For users, this means SQL is never truly portable across engines, and there is high risk of lock-in. We have solved this across Google engines with our shared and open-source dialect implementation, but lack of portability continues to be an industry-wide challenge.

In more recent years, the SQL functionality of Dremel has expanded greatly. The new shuffle architecture enabled support for joins, analytic functions and other complex queries. This has been driven both by Google's increasing need for more advanced analysis and by the demands from cloud users of BigQuery for feature parity with other familiar data warehouse products.

A similar journey away from SQL and back has happened in the open source world. Users outside Google had similar scale and cost challenges with increasing data sizes. Distributed file systems and MapReduce became popular with Hadoop, and a suite of other NoSQL systems followed. These users faced the same challenges with complexity and slow iteration. A similar pivot back

[3] https://github.com/google/zetasql

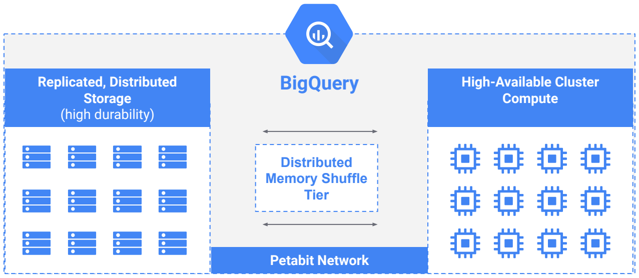

BigQuery
Replicated, Distributed Storage
(high durability)
Distributed Memory Shuffle Tier
High-Available Cluster Compute
Petabit Network

Figure 1: Disaggregated storage, memory, and compute

to SQL has happened, witnessed by the popularity of systems like HiveSQL[4], SparkSQL[5] and Presto.[6]

3. DISAGGREGATION

3.1 Disaggregated storage

Dremel was conceived at Google in 2006 as a "20 percent"[7] project by Andrey Gubarev. Initially, Dremel ran on a few hundred shared-nothing servers.[8] Each server kept a disjoint subset of the data on local disks. At the time, it seemed the best way to squeeze out maximal performance from an analytical system was by using dedicated hardware and direct-attached disks. As Dremel's workload grew, it became increasingly difficult to manage it on a small fleet of dedicated servers.

A major shift happened in early 2009. Dremel was migrated to a cluster management system called Borg [45]. (Borg was the first unified container-management system developed at Google, and a precursor to the open-source platform Kubernetes [11].) Moving to managed clusters was essential to accommodate the growing query workload and improve the utilization of the service. Yet, it exposed one of the challenges of using shared resources: the spindles used for Dremel's data were shared with other jobs. Consequently, we switched to a replicated storage organization, where a portion of each table was kept on three different local disks and managed by independent servers.

In combination, managed clusters and replicated storage contributed to a significant increase in Dremel's scalability and speed, and pushed its datasets into the range of petabyte-sized and trillion-row tables. Storing replicated data on local disks meant that storage and processing were coupled in an intimate way. This had a number of disadvantages: adding new features was hard because all algorithms needed to be replication-aware, the serving system could not be resized without shifting data around, scaling storage required adding servers and scaling CPU as well, and not least, the data was "locked up", i.e., it could not be accessed in any other way but via Dremel. All of these problems were solvable but the prospective solution we were heading toward was starting to look awkwardly similar to an existing core piece of infrastructure: Google's distributed file system, GFS [21].

Given the dramatic improvements in Google's storage and networking fabric, it was time to revisit the shared-nothing architecture. The obvious blocking issue was data access latency. In our first experiment with a GFS-based Dremel system, we saw an order-of-magnitude performance degradation. One issue was that scanning a table consisting of hundreds of thousands of tablets required opening as many files in GFS, which took multiple seconds, counting towards the query response time. Furthermore, the metadata format used by Dremel originally was designed for disk seeks as opposed to network roundtrips.

Harnessing query latency became an enduring challenge for Dremel engineers, which we cover in more detail in Section 7. It took a lot of fine-tuning of the storage format, metadata representation, query affinity, and prefetching to migrate Dremel to GFS. Eventually, Dremel on disaggregated storage outperformed the local-disk based system both in terms of latency and throughput for typical workloads.

In addition to liberating the data and reducing complexity, disaggregated storage had several other significant advantages. First, GFS was a fully-managed internal service, which improved the SLOs and robustness of Dremel. Second, the initial step of loading sharded tables from GFS onto Dremel server's local disks was eliminated. Third, it became easier to onboard other teams to the service since we did not need to resize our clusters in order to load their data. Another notch of scalability and robustness was gained once Google's file system was migrated from the single-master model in GFS to the distributed multi-master model in its successor, Colossus [31, 34].

3.2 Disaggregated memory

Shortly after the publication of the original paper, Dremel added support for distributed joins through a *shuffle* primitive. Inspired by the MapReduce shuffle implementation [19], Dremel's shuffle utilized local RAM and disk to store sorted intermediate results. However, the tight coupling between the compute nodes and the intermediate shuffle storage proved to be a scalability bottleneck:

1. With such colocation, it is not possible to efficiently mitigate the quadratic scaling characteristics of shuffle operations as the number of data producers and consumers grew.

2. The coupling inherently led to resource fragmentation and stranding and provides poor isolation. This became a major bottleneck in scalability and multi-tenancy as the service usage increased.

Continuing on the disaggregation path, we built a new disaggregated shuffle infrastructure using the Colossus distributed file system in 2012. This implementation encountered all of the challenges described in [38, 47]. After exploring alternatives, including a dedicated shuffle service, in 2014 we finally settled on the shuffle infrastructure which supported completely in-memory query execution [4]. In the new shuffle implementation, RAM and disk resources needed to store intermediate shuffle data were managed separately in a distributed transient storage system (see Figure 2).

Upon deployment, the in-memory shuffle implementation improved the Dremel query execution engine in several ways:

- Reduced the shuffle latency by an order of magnitude.

- Enabled an order of magnitude larger shuffles.

- Reduced the resource cost of the service by more than 20%.

The memory disaggregation in general, and the in-memory shuffle primitive in particular, significantly impacted the architecture

[4] https://hive.apache.org
[5] https://spark.apache.org/sql
[6] https://prestodb.io
[7] https://en.wikipedia.org/wiki/20%25_Project
[8] Shared-nothing was pioneered in the mid-1980s in the Teradata and Gamma projects as an architectural paradigm for parallel database systems based on a cluster of commodity computers with separate CPU, memory, and disks connected through a high-speed interconnect.

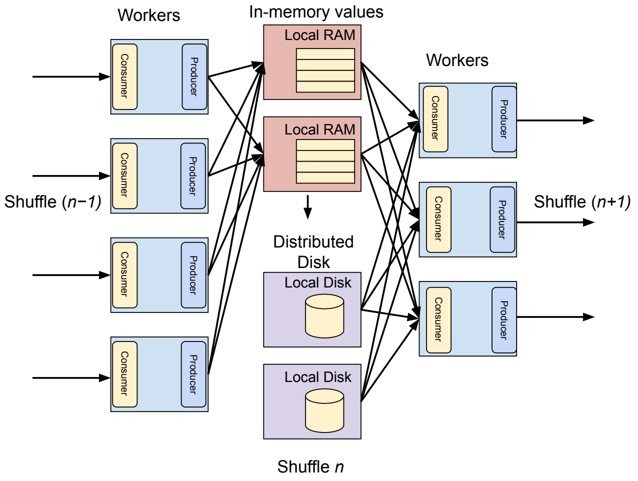


Figure 2: Disaggregated in-memory shuffle

of data analytics platforms such as Flume [12] and Google Cloud Dataflow.[9] Today, BigQuery continues to use the same disaggregated memory shuffle system where the disaggregated memory accounts for 80% of the total memory footprint of the service (see Figure 1). The same shuffle layer implementation and infrastructure also powers Google Cloud Dataflow [30].

3.3 Observations

Disaggregation proved to be a major trend in data management, as it decouples provisioning of different types of resources and enables better cost-performance and elasticity. Several aspects of disaggregation stand out:

- *Economies of scale*: The path of storage disaggregation went from RAID, SAN, distributed file systems to warehouse-scale computing [9].

- *Universality*: Storage disaggregation has been embraced by analytical and transactional systems alike, including Spanner [17], AWS Aurora [44], Snowflake [18], and Azure SQL Hyperscale [7]. Disaggregated flash is discussed in [10, 28].

- *Higher-level APIs*: Disaggregated resources are accessed via APIs at ever higher levels of abstraction. Storage access is far removed from the early block I/O APIs and includes access control, encryption at rest,[10] customer-managed encryption keys, load balancing, and metadata services (e.g., see [39]). Some data access APIs have built-in support for filtering and aggregation (see Section 6.1), which may be pushed all the way down into hardware (e.g., as done in Oracle SPARC M7 [5]).

- *Value-added repackaging*: Raw resources are packaged as services providing enhanced capabilities. Even if it is not practical to disaggregate a raw resource such as RAM for general-purpose use in a data management system, it may be cost-effective to factor it out as a value-added service, as exemplified by Dremel's shuffle.

4. IN SITU DATA ANALYSIS

In situ data processing refers to accessing data in its original place, without upfront data loading and transformation steps. In their prescient 2005 paper [22], Jim Gray et al. outlined a vision for scientific data management where a synthesis of databases and file systems enables searching petabyte-scale datasets within seconds. They saw a harbinger of this idea in the MapReduce approach pioneered by Google, and suggested that it would be generalized in the next decade. An explicit and standard data access layer with precise metadata was deemed a crucial ingredient for data independence.

Indeed, the data management community finds itself today in the middle of a transition from classical data warehouses to a data-lake-oriented architecture for analytics [1]. Three ingredients were called out as central to this transition: (a) consuming data from a variety of data sources, (b) eliminating traditional ETL-based data ingestion from an OLTP system to a data warehouse, and (c) enabling a variety of compute engines to operate on the data. We have observed each part of this transition in Dremel's decade-long history.

4.1 Dremel's evolution to in situ analysis

Dremel's initial design in 2006 was reminiscent of traditional DBMSs: explicit data loading was required, and the data was stored in a proprietary format, inaccessible to other tools. At the time, many tools at Google, including MapReduce, adopted a common record-oriented format, which consisted of sharded files in a distributed file system and a "schema" definition (Protocol Buffer record descriptor [37]) stored in the source code repository.

As part of migrating Dremel to GFS, we "open-sourced" our storage format within Google via a shared internal library. This format had two distinguishing properties: it was columnar and self-describing. Each file storing a data partition of a table also embedded precise metadata, which included the schema and derived information, such as the value ranges of columns. A self-describing storage format in GFS enabled interoperation between custom data transformation tools and SQL-based analytics. MapReduce jobs could run on columnar data, write out columnar results, and those results could be immediately queried via Dremel. Users no longer had to load data into their data warehouse. Any file they had in the distributed file system could effectively be part of their queryable data repository. The paradigms of MapReduce and parallel DBMSs turned out to be friends, not foes [43].

Having all of Dremel's data available in a shared distributed file system and encoded using a standard, open-sourced format created an environment in which many tools could develop and flourish. Over the past decade, the details of a number of these systems have been published, including Tenzing [16], PowerDrill [24], F1 [41, 40], and Procella [15]. All of these share data with Dremel in some way, which could be seen as redundant and self-competitive behavior on Google's part. However, the advances we have observed from this friendly competition as well as from collaboration and code sharing between the teams have far outweighed the costs of redundant engineering work. We believe a similar environment for innovation would have been impossible in the traditional world of proprietary data formats and siloed storage.

Over time we evolved our in situ approach in two complementary directions. First, we began adding file formats beyond our original columnar format. These included record-based formats such as Avro, CSV, and JSON. This expanded the range of data users could query using Dremel at the cost of increased I/O latency due to the need to read full records for most of these formats and the need to convert data on the fly for processing. We discovered that users were often willing to endure additional query latency to avoid the cost of re-encoding their data.

[9] https://cloud.google.com/dataflow

[10] https://cloud.google.com/bigquery/docs/encryption-at-rest

The second direction was expanding in situ analysis through federation. In some cases, including remote file systems such as Google Cloud Storage[11] and Google Drive,[12] we read the files directly. In other cases, including F1, MySQL, and BigTable, we read data through another engine's query API. In addition to expanding the universe of joinable data, federation allows Dremel to take advantage of the unique strengths of these other systems. For example, a lookup join which uses the row key in BigTable can be executed much more efficiently by reading only a subset of the rows rather than reading the entire table.

4.2 Drawbacks of in situ analysis

There were, however, important drawbacks to Dremel's in situ approach. First, users do not always want to or have the capability to manage their own data safely and securely. While this extra complexity in data governance was acceptable to some degree inside Google, it was not tolerable for many external customers. Second, in situ analysis means there is no opportunity to either optimize storage layout or compute statistics in the general case. In fact, a large percentage of Dremel queries are run over data seen for the first time. This makes many standard optimizations impossible. It is also impractical to run DML updates and deletes or DDL schema changes on standalone files.

These issues led to the creation of BigQuery Managed Storage[13] for cloud users as well as other managed solutions inside Google. We have observed that both managed and in situ data are expected in a successful big-data analytical solution. Users will have some data they self-manage in file systems, for various reasons, but that burden of self-management should not be forced on users when unnecessary or counter-productive. Complementary managed storage systems can provide the best of both worlds, mitigating the drawbacks encountered with in situ analysis.

Hybrid models that blend the features of in situ and managed storage were explored in NoDB [6] and Delta Lake.[14]

5. SERVERLESS COMPUTING

Dremel is one of the pioneers in providing an elastic, multi-tenant, and on-demand service, now widely referred to in the industry as *serverless*. In this section, we discuss how Dremel took a different approach from the industry, the core ideas that enabled the serverless architecture, and eventually how the industry adopted those ideas.

5.1 Serverless roots

Mostly following the pattern of database management systems, data warehouses such as IBM Netezza, Teradata, and Oracle products were deployed on dedicated servers at the time when Dremel was conceived. Big Data frameworks such as MapReduce and Hadoop used a more flexible deployment pattern, taking advantage of virtual machines and containers but still requiring single-tenant resource provisioning, i.e., a job per user.

It was evident that supporting interactive, low-latency queries and in situ analytics while scaling to thousands of internal users at Google at a low cost would only be possible if the service was multi-tenant and provided on-demand resource provisioning.

Initially, we took advantage of three core ideas to enable serverless analytics:

1. *Disaggregation*: The disaggregation of compute, storage, and memory allows on-demand scaling and sharing of compute independently from storage. Consequently, it allows the system to adapt to usage at a lower cost. As described in Section 3, Dremel started by decoupling disk storage from compute resources in 2009 and eventually added disaggregated memory in 2014.

2. *Fault Tolerance and Restartability*: Dremel's query execution was built based on the assumption that the underlying compute resources may be slow or unavailable, making the workers inherently unreliable. This assumption had strong implications for the query runtime and dispatching logic:

   - Each subtask within a query had to be deterministic and repeatable such that in case of failure, only a small fraction of the work needed to be restarted on another worker.
   - The query task dispatcher had to support dispatching multiple copies of the same task to alleviate unresponsive workers.

   Consequently, these mechanisms enabled the scheduling logic to easily adjust the amount of resources allocated to a query by cancelling and rescheduling subtasks.

3. *Virtual Scheduling Units*: Instead of relying on specific machine types and shapes, Dremel scheduling logic was designed to work with abstract units of compute and memory called *slots*. This was a well-suited model for the container-oriented Borg compute environment, which supported flexible resource allocation shapes. These virtual scheduling units allowed decoupling the scheduling and customer-visible resource allocation from the container and machine shapes and service deployment. Slots continue to be the core customer-visible concept of resource management in BigQuery.

These three ideas leveraged in the original Dremel paper became building blocks in many serverless data analytics systems. Disaggregation has been broadly adopted by industry and academia. Virtual resource units have been adopted by other providers such as Snowflake [18]. In many domains, the industry has converged on a data lake architecture, which uses elastic compute services to analyze data in cloud storage "on-demand". Additionally, many data warehouse services such as Presto, AWS Athena, and Snowflake have also adopted either on-demand analytics or automatic scaling as a key enabler for serverless analytics, leading many enterprises to adopt the cloud over on-premises systems.

5.2 Evolution of serverless architecture

Dremel continued to evolve its serverless capabilities, making them one of the key characteristics of Google BigQuery today. Some approaches in the original Dremel paper evolved into new ideas, described below, that took the serverless approach to the next level.

Centralized Scheduling. Dremel switched to centralized scheduling in 2012 which allowed more fine-grained resource allocation and opened the possibility for reservations, i.e., allocating a fraction of Dremel's processing capacity for specific customers. Centralized scheduling superseded the "dispatcher" from the original paper, which was responsible for resource distribution among queries in the intermediate servers (see Figure 3). The new scheduler uses the entire cluster state to make scheduling decisions which enables better utilization and isolation.

[11] https://cloud.google.com/bigquery/external-data-cloud-storage

[12] https://cloud.google.com/bigquery/external-data-drive

[13] https://cloud.google.com/bigquery/docs/reference/storage

[14] https://github.com/delta-io/delta

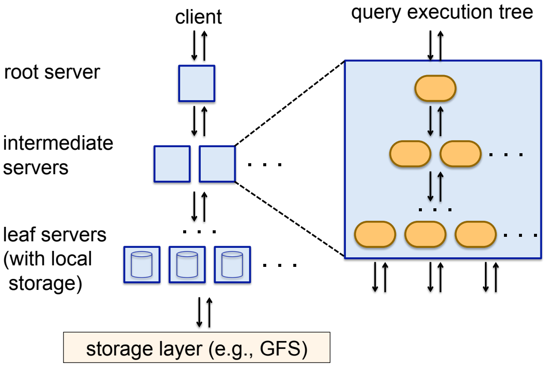
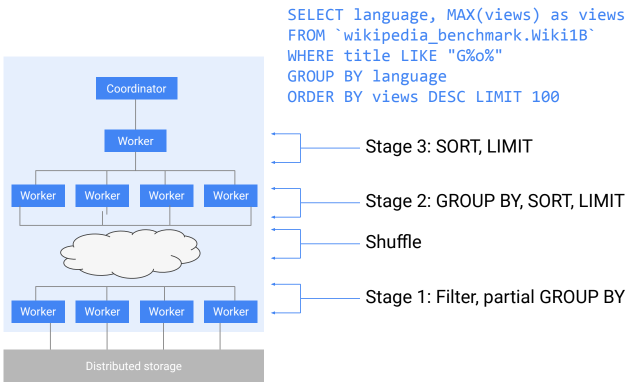


Figure 3: System architecture and execution inside a server node (Figure 7 in [32])



```
SELECT language, MAX(views) as views
FROM `wikipedia_benchmark.Wiki1B`
WHERE title LIKE "G%o%"
GROUP BY language
ORDER BY views DESC LIMIT 100
```

Figure 4: Shuffle-based execution plan

Shuffle Persistence Layer. Shuffle and distributed join functionality were introduced after the publication of the 2010 paper, as described in Section 3.2. After the initial shuffle implementation, the architecture evolved to allow decoupling scheduling and execution of different stages of the query. Using the result of shuffle as a checkpoint of the query execution state, the scheduler has the flexibility to dynamically preempt workers, reducing resource allocation to accommodate other workloads when compute resources are constrained.

Flexible Execution DAGs. The original paper described the system architecture shown in Figure 3. The fixed execution tree worked well for aggregations, but as Dremel evolved, the fixed tree was not ideal for more complex query plans. By migrating to the centralized scheduling and the shuffle persistence layer, the architecture changed in the following ways:

- The query coordinator is the first node receiving the query. It builds the query plan which could be a DAG of query execution trees (also known as *stages*), and then orchestrates the query execution with workers given to it by the scheduler.

- Workers are allocated as a pool without predefined structure. Once the coordinator decides on the shape of the execution DAG, it sends a ready-to-execute local query execution plan (tree) to the workers. Workers from the leaf stage read from the storage layer and write to the shuffle persistence layer, while workers from other stages read and write from/to the shuffle persistence layer. Once the entire query is finished, the final result is stored in the shuffle persistence layer, and the query coordinator then sends it to the client.

Consider the example in Figure 4, which illustrates the execution of a top-k query over a Wikipedia table. The query proceeds as follows:

- Workers from stage 1 (leaf) read the data from distributed storage, apply the filter, partially pre-aggregate data locally and then shuffle the data by hash partitioning on the language field.

- Since the data is shuffled by the aggregation key, the workers from stage 2 can do the final GROUP BY aggregation, then sort by a different key, truncate by the limit, and send the result to the next stage.

- In stage 3 there is just one worker; it reads the input from the shuffle persistence layer, does the final ordering and truncation, and writes results to the shuffle layer.

- The query coordinator reads the final 100 records from the shuffle persistence layer and sends them to the client.

Any Dremel query, such as the example presented above, can be executed on any number of workers, ranging from one to tens of thousands of workers. The shuffle persistence layer provides this flexibility.

Dynamic Query Execution. There are multiple optimizations that query engines can apply based on the shape of the data. For example, consider choosing the join strategy, e.g., broadcast vs hash join. Broadcast join does not need to shuffle data on the probe side of the join so it can be considerably faster, but broadcast only works if the build side is small enough to fit in memory.

Generally, it is difficult to obtain accurate cardinality estimates during query planning; it is well-known that errors propagate exponentially through joins [27]. Dremel has chosen a path where the query execution plan can dynamically change during runtime based on the statistics collected during query execution. This approach became possible with the shuffle persistence layer and centralized query orchestration by the query coordinator. In the case of broadcast-vs-hash join, Dremel will start with the hash join by shuffling data on both sides, but if one side finishes fast and is below a broadcast data size threshold, Dremel will cancel the second shuffle and execute a broadcast join instead.

6. COLUMNAR STORAGE FOR NESTED DATA

In the early 2000s, new application and data models emerged (often associated with the rise of Web 2.0), where instead of writing data into normalized relational storage, applications would write semistructured data with flexible schemas (e.g., logs). Many programming frameworks facilitated the use of semistructured data. XML was traditionally used for this purpose; JSON became popular due to its simplicity compared to XML. Google introduced Protocol Buffers [37], Facebook came up with Thrift [42], and the Hadoop community developed Avro.[15]

While the idea of columnar storage was well known, the Dremel paper spearheaded the use of columnar storage for semistructured data. Google used protocol buffers extensively in all of its applications—Search, GMail, Maps, YouTube, etc. Development of many open source columnar formats for nested data has followed: in 2013, Twitter and Cloudera announced the Parquet file format[16] citing the influence of the Dremel paper, Facebook and

[15] https://avro.apache.org

[16] https://parquet.apache.org

```
DocId: 10                r1
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

```
DocId: 20                r2
Name
  Url: 'http://C'
```

Figure 5: Two sample nested records and their schema (based on Figure 2 in [32])

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 6: Columnar representation of the data in Figure 5 showing repetition levels (r) and definition levels (d)

DocId

value	p
10	true
20	true

Name

len
3
1

Name.Url

value	p
http://A	true
http://B	true
	false
http://C	true

Name.Language

len
2
0
1
0

Name.Language.Code

value	p
en-us	true
en	true
en-gb	true

Name.Language.Country

value	p
us	true
	false
gb	true

Figure 7: Columnar representation of the data in Figure 5 showing length (len) and presence (p)

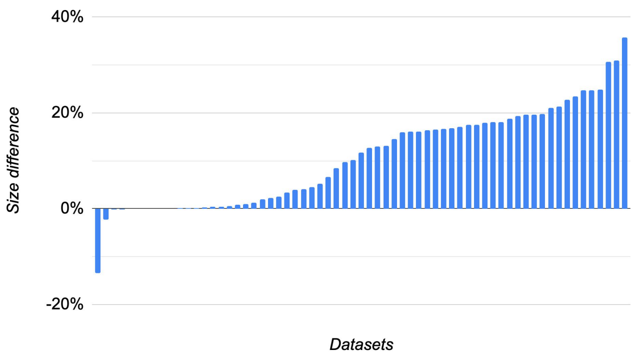


Figure 8: Percentage difference between repetition/definition and length/presence encodings. Higher bars correspond to smaller files with length/presence encoding.

Hortonworks came up with ORC[17], and in 2016 Apache Foundation announced Apache Arrow.[18]

All these formats support nested and repeated data, but they do it differently. The Dremel paper proposed the notion of *repetition* and *definition* levels to track repeated and optional fields, respectively. A detailed explanation of this encoding can be found in [33] but briefly, repetition level specifies for repeated values whether each ancestor record is appended into or starts a new value, and definition level specifies which ancestor records are absent when an optional field is absent. Parquet adopted this encoding.

ORC took a different approach tracking the lengths of repeated fields (i.e., number of occurrences within parent) and boolean attributes indicating the presence of optional fields. Arrow uses a similar approach to ORC but tracks repeated fields by their offsets (i.e., cumulative lengths). Using offsets facilitates direct access to array elements and makes sense for an in-memory format like Arrow, while storing lengths makes sense for an on-disk file format, as it compresses better.

To compare these approaches, consider the example in Figure 5. Encoding with repetition and definition levels is shown in Figure 6. Name.Language.Country is encoded as follows: its maximum repetition level is 2 because there are 2 repeated fields in its path, Name and Language. Its maximum definition level is 3 because there are 3 repeated or optional fields in its path, Name, Language, and Country. Let us start with record r_1. Repetition level is always 0 at the beginning of the record. Definition level is 3, because all 3 parts of the path are present in the first value 'us'. As we move to the next value, Country is now missing (depicted as NULL), but both Name and Language are still defined (containing Code 'en'), therefore definition level becomes 2. Repetition level tells us which repeated field in the path changed. It was Language (since Name remained the same), the 2nd repeated field, therefore repetition level is 2.

The length and presence encoding is shown in Figure 7. Name.Language lists how many times Language occurs in each successive Name record, i.e., 2, 0, 1, and 0 times, respectively, with a total of 3. Name.Language.Country contains the corresponding Country values from these 3 records.

There are tradeoffs between these approaches. The main design decision behind repetition and definition levels encoding was to encode all structure information within the column itself, so it can be accessed without reading ancestor fields. Indeed, the non-leaf nodes are not even explicitly stored. However, this scheme leads to redundant data storage, since each child repeats the same information about the structure of common ancestors. The deeper and wider the structure of the message, the more redundancy is introduced. Analyzing both encodings over 65 internal Google datasets which featured deep and wide messages, the length/presence encoding consistently resulted in smaller file sizes, being on average 13% smaller than the repetition/definition level encoding (see Figure 8).

[17] https://orc.apache.org

[18] https://arrow.apache.org

Algorithms for processing analytical queries on top of these encodings are different. With repetition/definition levels it is sufficient to only read the column being queried, as it has all required information. In 2014, we published efficient algorithms [3] for Compute, Filter and Aggregate that work with this encoding. With length/presence encoding, it is also necessary to read all ancestors of the column. This incurs additional I/O, and while ancestor columns usually are very small, they could require extra disk seeks. Also, algorithms for detecting array boundaries require looking at counts at multiple levels. Quantifying the tradeoffs between these encodings is an area of future research.

In 2014, we began migration of the storage to an improved columnar format, Capacitor [35]. It was built on the foundation described in the original Dremel paper, and added many new features. Some of those enhancements are described below.

6.1 Embedded evaluation

To make filtering as efficient as possible, we embedded it directly into the Capacitor data access library. The library includes a mini query processor which evaluates SQL predicates. This design choice allowed us to have efficient filter support in all data management applications using Capacitor—not just Dremel but also F1, Procella, Flume, MapReduce, and BigQuery's Storage API.[19] For example, the Storage API allows specifying SQL predicates as strings as part of read options.

Capacitor uses a number of techniques to make filtering efficient:

- *Partition and predicate pruning*: Various statistics are maintained about the values in each column. They are used both to eliminate partitions that are guaranteed to not contain any matching rows, and to simplify the filter by removing tautologies. For example, the predicate EXTRACT(YEAR FROM date) = 2020 is first rewritten as date BETWEEN '2020-01-01' AND '2020-12-31' and is used to eliminate all partitions outside of this date range. A more complex example is ST_DISTANCE(geo, constant_geo) $<$ 100, returning only values which are within 100 meters of a given constant object. In this case, more advanced statistics are used to compare S2 covering[20] of constant_geo and union of S2 coverings of all values in the file.

- *Vectorization*: Columnar storage lends itself to columnar block-oriented vectorized processing. Capacitor's embedded query evaluator uses most of the techniques described in [2].

- *Skip-indexes*: Filter predicates used in internal and external BigQuery workloads tend to be very selective. Figure 9 shows that about 15% of queries return no data (have selectivity 0), about 25% of queries return less than 0.01% of the data, and about 50% of queries return less than 1% of the data. High selectivity requires a fast implementation of skipping to jump over the records where the predicate evaluated to false. To do that, at write time Capacitor combines column values into segments, which are compressed individually. The column header contains an index with offsets pointing to the beginning of each segment. When the filter is very selective, Capacitor uses this index to skip segments that have no hits, avoiding their decompression.

- *Predicate reordering*: While the optimal algorithm for predicate reordering in a filter is known [25], it relies on the a priori knowledge of each predicate's selectivity and cost, which are hard to estimate. Capacitor uses a number of heuristics to make filter reordering decisions, which take into account dictionary usage, unique value cardinality, NULL density, and expression complexity. For example, consider a filter $p(x)$ AND $q(y)$, where x has no dictionary encoding and many unique values, while y has a dictionary and only few unique values. In this case, it is better to evaluate predicate $q(y)$ followed by $p(x)$, even if $q(y)$ is a more complex expression than $p(x)$, since $q(y)$ will only be evaluated over a small number of dictionary values.

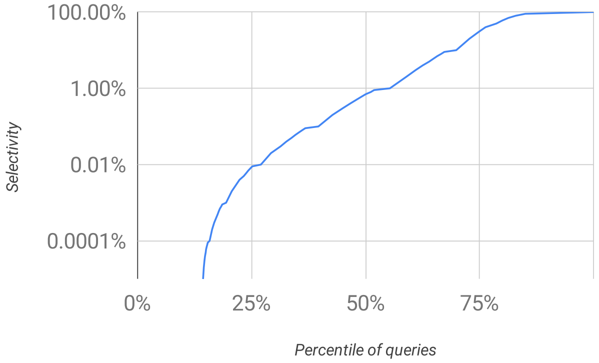


Figure 9: Distribution of filter selectivities in queries

6.2 Row reordering

Capacitor uses several standard techniques to encode values, including dictionary and run-length encodings (RLE). RLE in particular is very sensitive to row ordering. Usually, row order in the table does not have significance, so Capacitor is free to permute rows to improve RLE effectiveness. Let us illustrate using an example of three columns in Figure 10. Encoding this data with RLE using the existing order would be suboptimal since all run lengths would be 1, resulting in 21 runs. However, if the input rows are reshuffled as shown in Figure 10, we obtain a total of 9 runs. This permutation is optimal for the given input and produces better results than lexicographical sort by any combination of columns.

Unfortunately, finding the optimal solution is an NP-complete problem [29], i.e., impractical even for a modest number of input rows, let alone for billions of rows. To further complicate matters, not all columns are born equal: short RLE runs give more benefit for long strings than longer runs on small integer columns. Finally, we must take into account actual usage: some columns are more likely to be selected in queries than others, and some columns are more likely to be used as filters in WHERE clauses. Capacitor's row reordering algorithm uses sampling and heuristics to build an approximate model. Its details are beyond the scope of this paper.

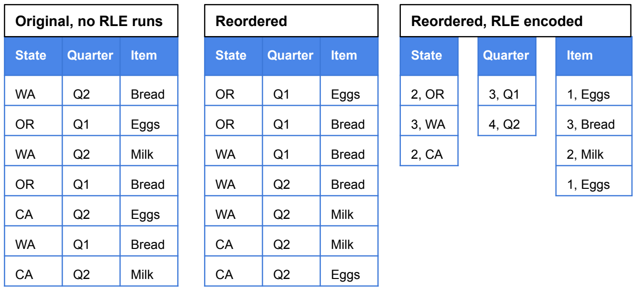
Row reordering works surprisingly well in practice. Figure 11 shows size differences when enabling reordering on 40 internal Google datasets. Overall saving was 17%, with some datasets reaching up to 40% and one up to 75%.

6.3 More complex schemas

Protocol buffers allow defining recursive message schemas. This is useful to model many common data structures. For example, a tree can be defined as

[19] https://cloud.google.com/bigquery/docs/reference/storage

[20] https://s2geometry.io/devguide/s2cell_hierarchy.html



Original, no RLE runs

State	Quarter	Item
WA	Q2	Bread
OR	Q1	Eggs
WA	Q2	Milk
OR	Q1	Bread
CA	Q2	Eggs
WA	Q1	Bread
CA	Q2	Milk

Reordered

State	Quarter	Item
OR	Q1	Eggs
OR	Q1	Bread
WA	Q1	Bread
WA	Q2	Bread
WA	Q2	Milk
CA	Q2	Milk
CA	Q2	Eggs

Reordered, RLE encoded

State	Quarter	Item
2, OR	3, Q1	1, Eggs
3, WA	4, Q2	3, Bread
2, CA		2, Milk
		1, Eggs

Figure 10: Example of row reordering

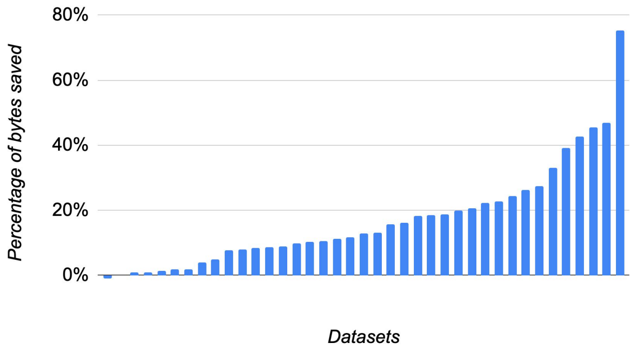


Figure 11: Impact of row reordering for diverse datasets

```
message Node {
  optional Payload value;
  repeated Node nodes;
}
```

One challenge is that the maximal recursion depth used in a given dataset is not known ahead of time. Dremel did not originally support recursive messages of arbitrary depth. Capacitor added such support.

Another new challenge is supporting messages without a strict schema. This is typical with JSON messages or with XML without XSD. Protocol Buffers allow it through extensions[21] and using proto3 with 'Any' message type.[22] The main challenge is not just that new columns can appear in any row, but also that a column with the same name can have a varying type from message to message.

Capacitor has only partially solved the problems outlined in this section. Efficient storage and access to heterogeneous columnar data remains an area of active research.

7. INTERACTIVE QUERY LATENCY OVER BIG DATA

The design principles introduced earlier (disaggregation, in situ processing, and serverless) tend to be counterproductive to building a system for interactive query latency. Conventional wisdom has it that colocating processing with data reduces data access latency, which goes counter to disaggregation. Optimizing the storage layout of the data runs counter to in situ processing. Dedicated machines should be more performant than shared serverless machine resources.

In this section, we discuss some of the latency-reducing techniques implemented in Dremel to achieve interactive query processing speeds, beyond using columnar storage.

Stand-by server pool. With a distributed SQL execution engine, it is possible to bring up a system and have it ready to process queries as soon as they are submitted. This eliminates the machine allocation, binary copying, and binary startup latencies that existed when users wrote their own MapReduce or Sawzall jobs.

Speculative execution. When a query is processed by hundreds of machines, the slowest worker can be an order of magnitude slower than the average. Without any remediation, the end effect is that the user's end-to-end query latency is an order of magnitude higher. To address that problem, Dremel breaks the query into thousands of small tasks, where each worker can pick up tasks as they are completed. In this way, slow machines process fewer tasks and fast machines process more tasks. Moreover, to fight long tail latency at the end of query execution, Dremel can issue duplicate tasks for stragglers, bringing the total latency down. Thus performance becomes a function of total available resources and not of the slowest component.

Multi-level execution trees. Being able to use hundreds of machines to process a single query in a few seconds requires coordination. Dremel solved this using a tree architecture with a root server on top of intermediate servers on top of leaf servers. Execution flows from the root to the leaves and back (see Figure 3). This model was originally borrowed from Google's Search. It worked well for parallelizing both the dispatching of requests and the assembly of query results.

Column-oriented schema representation. Dremel's storage format was designed to be self-describing, i.e., data partitions store embedded schemas. The schemas used at Google often contain thousands of fields. Parsing a complete schema might take longer than reading and processing the data columns from a partition. To address that, Dremel's internal schema representation was itself stored in a columnar format.

Balancing CPU and IO with lightweight compression. Using a columnar format makes compression more effective because similar values (all values of a single column) are stored sequentially. This means fewer bytes have to be read from the storage layer, which in turn reduces query latency. On the other hand, decompressing data requires CPU cycles, so the more involved the compression, the higher the CPU cost. The key is to pick a compression scheme that balances data size reduction with CPU decompression cost so that neither CPU nor IO becomes the bottleneck.

Approximate results. Many analyses do not require 100% accuracy, so providing approximation algorithms for handling top-k and count-distinct can reduce latency. Dremel uses one-pass algorithms that work well with the multi-level execution tree architecture (e.g., [14, 26]). In addition Dremel allows users to specify what percentage of data to process before returning the result. Due to the straggler effect, returning after 98% of the data has been processed has been shown to improve latency by 2-3× (see Figure 12).

Query latency tiers. To achieve high utilization in a shared server pool, Dremel had to natively support multiple users issuing multiple queries simultaneously. With a wide range of data sizes, some queries can be sub-second while others take tens of seconds. To ensure that "small" queries remain fast and do not get blocked by users with "large" queries, Dremel used a dispatcher

[21] https://developers.google.com/protocol-buffers/docs/proto#extensions

[22] https://developers.google.com/protocol-buffers/docs/proto3#any

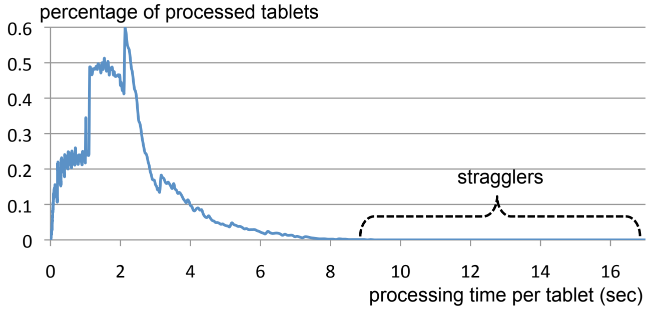


Figure 12: Stragglers (Figure 14 in [32])

on the intermediate servers to schedule resources fairly. The dispatcher needed to be able to preempt the processing of parts of a query to allow a new user's query to be processed to avoid the situation where previous users are monopolizing resources by running hundreds of concurrent queries. Even queries from a single user may require different priorities, such as queries backing interactive dashboards versus queries performing daily ETL pipeline jobs.

Reuse of file operations. Processing hundreds of thousands of files for a query in a few seconds places an enormous load on the distributed file system. This can actually become the bottleneck for achieving low latency as thousands of Dremel workers send requests to the file system master(s) for metadata and to the chunk servers for open and read operations. Dremel solved this with a few techniques. The most important one was to reuse metadata obtained from the file system by fetching it in a batch at the root server and passing it through the execution tree to the leaf servers for data reads. Another technique was to create larger files so the same table can be represented by fewer files, resulting in reduced file metadata operations.

Guaranteed capacity. The concept of reservations introduced with the centralized scheduler in Section 5 also helped with latency. For example, a customer could reserve some capacity and use that capacity only for latency-sensitive workloads. When guaranteed capacity is underutilized, these resources are available for others to use, but when requested those resources are immediately granted to the customer. Dremel workers use a custom thread scheduler which instantly reallocates CPUs to reserved workloads and pauses nonreserved workloads.

Adaptive query scaling. Flexible execution DAGs described in Section 5 were an important part of improving latency given growing, diverse workloads. The execution DAG can be built individually for each query based on the query plan. Consider a global aggregation, e.g., COUNT or SUM: with a fixed aggregation tree, such a query had to do multiple hops through intermediate levels, but with flexible DAGs there is no need to have more than two aggregation levels—the leaf level aggregates the input and produces one record per file and the top level does the final aggregation. In contrast, consider a top-k query, i.e., ORDER BY ... LIMIT. Each worker in the leaf stage produces many records. Doing the final aggregation on the single node with a large number of inputs is going to be a bottleneck. Therefore, to process this query Dremel dynamically builds an aggregation tree whose depth depends on the size of the input.

8. CONCLUSIONS

Looking back over the last decade, we are proud of how much the original Dremel paper got right and humbled by the things we got wrong.

Among the things the original paper got right, architectural choices including disaggregated compute and storage, on-demand serverless execution, and columnar storage for nested, repeated and semistructured data have all become standard industry best practices. In situ data analysis has been recognized as essential by the Data Lakes movement, and the wisdom of analyzing data in place whenever possible is now taken for granted. There has been a resurgence in the use of SQL for Big Data applications, reversing the trend toward exclusively NoSQL tools. And, of course, the possibility of achieving interactive speeds over very large datasets is now not only accepted, it is expected.

Things we got wrong or missed in the original paper include the need for a reliable and fast shuffle layer rather than a static aggregation tree, the importance of providing a managed storage option in addition to in situ analysis, and the need for rigorous SQL language design that honors existing standards and expected patterns.

On balance, we feel the Dremel paper has been an important contribution to our industry. We are pleased that it has stood the test of time.

9. ACKNOWLEDGMENTS

Far too many people have contributed to Dremel since 2006 for us to list them all in this paper. We would like to specifically acknowledge significant code contributions made by the following 274 engineers:[23] Jon Adams, Abdullah Alamoudi, Sonya Alexandrova, Tom Alsberg, Matt Armstrong, Ahmed Ayad, Nikolay Baginskyy, Paul Baker, Alex Balikov, Adrian Baras, Leonid Baraz, Mark Barolak, Max Barysau, Dmitry Belenko, Samuel Benzaquen, Alexander Bernauer, Alex Blyzniuchenko, Jeffrey Borwey, Eli Brandt, Andrew Brook, Elliott Brossard, JD Browne, Jim Caputo, Garrett Casto, James Chacon, Cissy Chen, Sean Chen, Shuo Chen, Xi Cheng, Bruce Chhay, Stanislav Chiknavaryan, Kang Kai Chow, Josef Cibulka, Craig Citro, Jeremy Condit, Ezra Cooper, Jason Cunningham, Steven Dai, Chris DeForeest, Hua Deng, Mingge Deng, Michael Diamond, Jojo Dijamco, Nicholas Doyle, Priyam Dutta, Pavan Edara, Tamer Eldeeb, Michael Entin, Greg Falcon, Nova Fallen, Viktor Farkas, Omid Fatemieh, Anton Fedorov, Eric Feiveson, Xandra Ferreron, Patrik Fimml, Amanda Fitch, Peter Foley, Jonathan Forbes, Drew Galbraith, Gabriel Gambetta, Charlie Garrett, Piotr Gawryluk, Qingtao Geng, Michelle Goodstein, Roman Govsheev, Vlad Grachev, Matt Gruchacz, Carmi Grushko, Teng Gu, Qing Guo, Abhishek Gupta, Josh Haberman, Mateusz Haligowski, Anastasia Han, Danielle Hanks, Michael Hanselmann, Jian He, Xi He, Jeff Heidel, Chris Hill, James Hill, Andrew Hitchcock, Steve Holder, Seth Hollyman, Amir Hormati, Thibaud Hottelier, Adam Iwanicki, Daniel Jasper, Dennis Jenkins, Shuai Jiang, Wenwei Jiang, Jeff Jolma, Prahlad Joshi, Navin Joy, Kenneth Jung, Heikki Kallasjoki, Daria Kashcha, Abhishek Kashyap, Vojin Katic, Andrei Khlyzov, Vladimir Khodel, Rohan Khot, Yulia Khudiakova, Danny Kitt, Nick Kline, Alexey Klishin, Kurt Kluever, Eugene Koblov, Choden Konigsmark, Leonid Konyaev, Micah Kornfield, Basha Kovacova, Andy Kreling, Matthew Kulukundis, Ju-yi Kuo, Younghee Kwon, Meir Lakhovsky, Alex Lamana, Byron Landry, Jeremie Le Hen, Alexey Leonov-Vendrovskiy, Caleb Levine, Bigang Li, Jing Li, Sha Li, Vitalii Li, Weizheng (Wilson) Li, Yuan Li, Vlad Lifliand, Benjamin Liles, Justin Lin, Deen Liu, Kevin Liu,

[23] Fittingly, this list was obtained via a top-k Dremel query that does a left outer join between Google's source code repository and the personnel database. We manually filled in the names of former Googlers whose personal information was erased due to data protection regulations.

Sasa Ljuboje, Samad Lotia, Siyao Lu, Grzegorz Lukasik, Mariëtte Luke, Leonid Lyakhovitskiy, Adam Lydick, Changming Ma, Yunxiao Ma, Kurt Maegerle, Uladzimir Makaranka, Michał Mamczur, Allen Mathew, Luca Mattiello, Chris McBride, Rossi McCall, Brandon McCarthy, Jeff McGovern, Chris Meyers, Michael Moursalimov, Cheng Miezianko, Ashish Misra, Martin Mladenovski, Nils Molina, David Morgenthaler, Victor Mota, Siddartha Naidu, Jeff Namkung, Gurpreet Nanda, Aryan Naraghi, Ahmed Nasr, Nhan Nguyen, Mark Nichols, Ning Rogers, Vadim Suvorov, Murat Ozturk, Pavel Padinker, Xuejian Pan, Iulia Parasca, Thomas Park, Anand Patel, Boris Pavacic, Alan Pearson, Gleb Peregud, Francis Perron, Cristian Petrescu Prahova, Denis Petushkov, John Phillpot, Ovidiu Platon, Evgeny Podlepaev, Steve Poloni, Iustin Pop, Jiaming Qiu, Jeff Quinlan-Galper, Makoto Raku, Marte Ramírez Ortegón, Alex Raschepkin, Chris Reimer, Erika Rice Scherpelz, Jason Roselander, Gennadiy Rozental, Julius Rus, Kwanho Ryu, Girishkumar Sabhnani, Surbhi Sah, Purujit Saha, Subhadeep Samantaray, Prathamesh Sancheti, Niranjan Sathe, Corwin Schillig, Drew Schmitt, Hani Semaan, Nitin Sharma, Michael Sheldon, Vachan Shetty, John Shumway, Jeremy Smith, Alexander Smolyanov, Andy Soffer, Rafal Sokolowski, Aaron Spangler, Piotr Stanczyk, Pete Steijn, Josh Stein, Paweł Stradomski, Sujay Krishna Suresh, Aleksandras Surna, Ian Talarico, Li Tan, Jonah Tang, Yiru Tang, Scott Tao, Tiho Tarnavski, Alex Thomson, Jordan Tigani, John Tribe, Eric Trottier, Sean Trowbridge, Natalie Truong, Jason Uanon, Andrey Ulanov, Ezra Umen, Mohsen Vakilian, Ela Verman, Sreeni Viswanadha, Matt Wachowski, Brad Walker, Carl Walsh, Aaron Wang, Matthew Weaver, Matthew Wesley, Judson Wilson, Adrian Wisniewski, Hyrum Wright, Jiaxun Wu, Liangshou Wu, Zhiting Xu, Stepan Yakovlev, Xintian Yang, Yi Yang, Jerry Ye, Zihao Ye, Steve Yen, Theodora Yeung, Lisa Yin, Jonathan Yu, Magda Zakrzewska, Dylan Zehr, Bei Zhang, Bowen Zhang, Hua Zhang, Lu Zhang, Tracy Zhang, Xiaoyi Zhang, Yun Zhang, Yaxin Zheng, Mingyu Zhong, Alice Zhu, Ian Zhu, and Tomasz Zielonka.

Many others have provided organizational or thought leadership, user feedback, and improvement suggestions. We recognize and appreciate all these contributions.

We are grateful to the VLDB Endowment Awards Committee for selecting our 2010 paper. We thank Phil Bernstein for his comments on an earlier draft of this paper.

10. REFERENCES

[1] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, et al. The Seattle Report on Database Research. *ACM SIGMOD Record*, 48(4), 2020.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

[3] F. N. Afrati, D. Delorey, M. Pasumansky, and J. D. Ullman. Storing and Querying Tree-Structured Records in Dremel. *PVLDB*, 7(12), 2014.

[4] H. Ahmadi. In-memory Query Execution in Google BigQuery. Google Cloud Blog, Aug 2016.

[5] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, et al. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 35(2), 2015.

[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[7] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, et al. Socrates: The New SQL Server in the Cloud. In *SIGMOD*, 2019.

[8] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL System. In *SIGMOD*, 2017.

[9] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[10] P. A. Bernstein, C. W. Reid, and S. Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Queue*, 14(1), 2016.

[12] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, 2010.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2011.

[14] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Intl. Colloquium on Automata, Languages, and Programming*. Springer, 2002.

[15] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, et al. Procella: Unifying Serving and Analytical Data at YouTube. *PVLDB*, 12(12), 2019.

[16] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: a SQL Implementation on the MapReduce Framework. *PVLDB*, 4(12), 2011.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[18] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The Snowflake Elastic Data Warehouse. In *SIGMOD*, 2016.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[20] M. J. Franklin. Technical Perspective – Data Analysis at Astonishing Speed. *Commun. ACM*, 54(6):113, 2011.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[22] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4), 2005.

[23] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB*, 7(12), 2014.

[24] A. Hall, O. Bachmann, R. Bussow, S. Ganceanu, and M. Nunkesser. Processing a Trillion Cells per Mouse Click. *PVLDB*, 5(11), 2012.

[25] M. Z. Hanani. An Optimal Evaluation of Boolean Expressions in an Online Query System. *Commun. ACM*, 20(5), 1977.

[26] S. Heule, M. Nunkesser, and A. Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *EDBT*, 2013.

[27] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, 1991.

[28] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proc. of the 11th European Conf. on Computer Systems*, 2016.

[29] D. Lemire, O. Kaser, and E. Gutarra. Reordering Rows for Better Compression: Beyond the Lexicographic Order. *ACM Trans. on Database Systems (TODS)*, 37(3), 2012.

[30] T. Marian, M. Dvorský, A. Kumar, and S. Sokolenko. Introducing Cloud Dataflow Shuffle: For up to 5× Performance Improvement in Data Analytic Pipelines. Google Cloud Blog, June 2017.

[31] M. K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *Queue*, 7(7), 2009.

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

[33] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Commun. ACM*, 54(6), 2011.

[34] C. Metz. Google Remakes Online Empire with 'Colossus'. *Wired [Online]. Available: http://www.wired.com/2012/07/google-colossus/*, 2012.

[35] M. Pasumansky. Inside Capacitor, BigQuery's Next-Generation Columnar Storage Format. Google Cloud Blog, Apr 2016.

[36] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming Journal*, 13:277–298, 2005.

[37] Protocol Buffers: Developer Guide. Available at http://code.google.com/apis/protocolbuffers/docs/overview.html.

[38] S. Qiao, A. Nicoara, J. Sun, M. Friedman, H. Patel, and J. Ekanayake. Hyper Dimension Shuffle: Efficient Data Repartition at Petabyte Scale in SCOPE. *PVLDB*, 12(10), 2019.

[39] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, et al. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *SIGMOD*, 2017.

[40] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, et al. F1 Query: Declarative Querying at Scale. *PVLDB*, 11(12), 2018.

[41] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. O. K. Littlefield, D. Menestrina, S. E. J. Cieslewicz, I. Rae, et al. F1: A Distributed SQL Database that Scales. *PVLDB*, 6(11), 2013.

[42] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper*, 5(8), 2007.

[43] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Commun. ACM*, 53(1), 2010.

[44] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*, 2017.

[45] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proc. of the 10th European Conf. on Computer Systems*, 2015.

[46] J. Yang, I. Rae, J. Xu, J. Shute, Z. Yuan, K. Lau, Q. Zeng, X. Zhao, J. Ma, Z. Chen, Y. Gao, Q. Dong, J. Zhou, J. Wood, G. Graefe, J. Naughton, and J. Cieslewicz. F1 Lightning: HTAP as a Service. *PVLDB*, 13(12), 2020.

[47] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *Proc. of the 13th EuroSys Conf.*, 2018.