F1 Query: Declarative Querying at Scale

Bart Samwel John Cieslewicz Ben Handy Jason Govig Petros Venetis
Chanjun Yang Keith Peters Jeff Shute Daniel Tenedorio Himani Apte
Felix Weigel David Wilhite Jiacheng Yang Jun Xu Jiexing Li Zhan Yuan
Craig Chasseur Qiang Zeng Ian Rae Anurag Biyani Andrew Harn Yang Xia
Andrey Gubichev Amr El-Helw Orri Erling Zhepeng Yan Mohan Yang
Yiqun Wei Thanh Do Colin Zheng Goetz Graefe Somayeh Sardashti
Ahmed M. Aly Divy Agrawal Ashish Gupta Shiv Venkataraman
Google LLC
f1-query-paper@google.com

ABSTRACT

F1 Query is a stand-alone, federated query processing platform that executes SQL queries against data stored in different file-based formats as well as different storage systems at Google (e.g., Bigtable, Spanner, Google Spreadsheets, etc.). F1 Query eliminates the need to maintain the traditional distinction between different types of data processing workloads by simultaneously supporting: (i) OLTP-style point queries that affect only a few records; (ii) low-latency OLAP querying of large amounts of data; and (iii) large ETL pipelines. F1 Query has also significantly reduced the need for developing hard-coded data processing pipelines by enabling declarative queries integrated with custom business logic. F1 Query satisfies key requirements that are highly desirable within Google: (i) it provides a unified view over data that is fragmented and distributed over multiple data sources; (ii) it leverages datacenter resources for performant query processing with high throughput and low latency; (iii) it provides high scalability for large data sizes by increasing computational parallelism; and (iv) it is extensible and uses innovative approaches to integrate complex business logic in declarative query processing. This paper presents the end-to-end design of F1 Query. Evolved out of F1, the distributed database originally built to manage Google's advertising data, F1 Query has been in production for multiple years at Google and serves the querying needs of a large number of users and systems.

PVLDB Reference Format:
B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, Z. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. M. Aly, D. Agrawal, A. Gupta, and S. Venkataraman. F1 Query: Declarative Querying at Scale. *PVLDB*, 11 (12): 1835-1848, 2018.
DOI: https://doi.org/10.14778/3229863.3229871

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229871

1. INTRODUCTION

The data processing and analysis use cases in large organizations like Google exhibit diverse requirements in data sizes, latency, data sources and sinks, freshness, and the need for custom business logic. As a result, many data processing systems focus on a particular slice of this requirements space, for instance on either transactional-style queries, medium-sized OLAP queries, or huge Extract-Transform-Load (ETL) pipelines. Some systems are highly extensible, while others are not. Some systems function mostly as a closed silo, while others can easily pull in data from other sources. Some systems query live data, but others must ingest data before being able to query it efficiently.

In this paper, we present F1 Query, an SQL query engine that is unique not because of its focus on doing one thing well, but instead because it aims to cover all corners of the requirements space for enterprise data processing and analysis. F1 Query effectively blurs the traditional distinction between transactional, interactive, and batch-processing workloads, covering many use cases by supporting: (i) OLTP point queries that affect only a few records, (ii) low-latency OLAP querying of large amounts of data, and (iii) large ETL pipelines transforming and blending data from different sources into new tables supporting complex analysis and reporting workloads. F1 Query has also significantly reduced the need for developing hard-coded data processing pipelines, by enabling declarative queries integrated with custom business logic. As such, F1 is a one-size-fits-all querying system that can support the vast majority of use cases for enterprise data processing and analysis.

F1 Query has evolved from F1 [55], a distributed relational database for managing revenue-critical advertising data within Google, which included a storage layer as well as an engine for processing SQL queries. In its early stages, this engine executed SQL queries against data stored in only two data sources: Spanner [23, 55] and Mesa [38], one of Google's analytical data warehouses. Today, F1 Query runs as a stand-alone, federated query processing platform to execute declarative queries against data stored in different file-based formats as well as different remote storage systems (e.g., Google Spreadsheets, Bigtable [20]). F1 Query has become the query engine of choice for numerous critical applications including Advertising, Shopping, Analytics, and Payments. The driving force behind this momentum comes from F1 Query's flexibility, enabling use cases large and small, with simple or highly customized business logic, and across whichever data sources the data resides in. We note that in many ways, F1 Query

re-implements functionality that is already present in commercial DBMS solutions. It also shares design aspects with Dremel [51], a Google query engine optimized for analytical queries. The main innovation that F1 Query brings in the technology arena is how it combines all of these ideas, showing that in modern datacenter architecture and software stacks, it is possible to fully disaggregate query processing from data storage, and to serve nearly all use cases with that approach.

In this paper, we discuss the overall design of F1 Query. The following key requirements have influenced its overall architecture:

Data Fragmentation. Google has many options for data management that cater to a wide range of use cases with often conflicting requirements, including replication, latency, consistency, etc. As a result, the underlying data for even a single application is often fragmented across several storage systems, some stored in a relational DBMS engine like Spanner storage, some in key-value stores like Bigtable, and some stored as files in a variety of formats on distributed file systems. F1 Query satisfies the need for analysis of data across all such storage systems, providing a unified view of data fragmented across individual silos.

Datacenter Architecture. F1 Query was built for datacenters instead of individual servers or tightly-coupled clusters. This design abstraction fundamentally differs from classical *shared nothing* [57] database management systems that attempt to keep computation and processing of data localized where the data resides at all times. Furthermore, the classical paradigm tightly couples the database storage subsystem with the query processing layer, often sharing memory management, storage layout, etc. In contrast, F1 Query decouples database storage from query processing, and as a result, it can serve as an engine for all data in the datacenter. Advances in datacenter networking at Google [56] largely remove the throughput and latency differential in accessing local versus remote data, at least for data that resides in secondary storage. In our context, local disks are neither a point of contention nor a throughput bottleneck, since all the data is distributed in small chunks across the Colossus File System (the successor to the Google File System [33]). Similarly, remote data management services such as Spanner are widely distributed and less sensitive to contention when subjected to balanced access patterns. In spite of these technological advances, latency for requests to the underlying data sources is subject to high variance even in a controlled datacenter environment [25]. Mitigating this variability is one of the major challenges that F1 Query addresses.

Scalability. Client needs vary widely, not only in the sizes of datasets being processed, but also in latency and reliability requirements, and allowable resource cost. In F1 Query, short queries are executed on a single node, while larger queries are executed in a low-overhead distributed execution mode with no checkpointing and limited reliability guarantees. The largest queries are run in a reliable batch-oriented execution mode that uses the MapReduce framework [26]. Within each of these modes, F1 Query mitigates high latencies for large data sizes by increasing the computational parallelism used for query processing.

Extensibility. Clients should be able to use F1 Query for any data processing need, including those not easily expressible in SQL or requiring access to data in new formats. To meet this need, F1 Query is highly extensible. It supports user-defined functions (UDFs), user-defined aggregate functions (UDAs), and table-valued functions (TVFs) to integrate complex business logic written in native code into query execution.

In the rest of this paper, we present the end-to-end design of F1 Query. We first provide an overview of F1 Query architecture in Section 2. We then dive into its execution kernel and the *interactive* execution modes in Section 3, and the MapReduce-based *batch* execution mode in Section 4. In Section 5, we describe the F1 Query optimizer, followed by Section 6 that covers the various extensibility options in F1. Section 7 covers the advanced topics of execution cliff avoidance and structured data handling in F1. We present production metrics in Section 8, the related work in section 9, and some concluding remarks in Section 10.

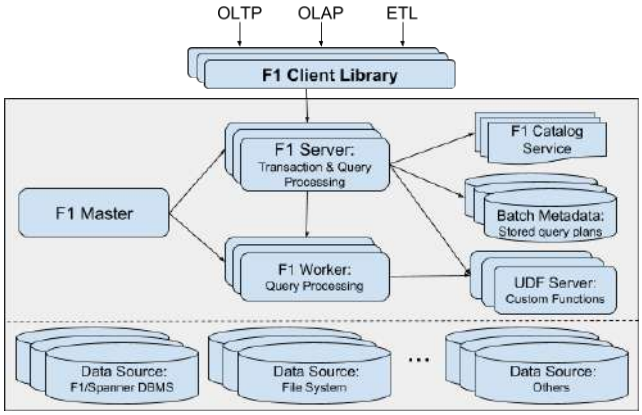

Figure 1: Overview of F1 Federated Query Processing Platform

2. OVERVIEW OF F1 QUERY

Architecture. F1 Query is a federated query engine that supports all OLTP, OLAP, and ETL workloads. Figure 1 depicts the basic architecture and communication among components within a single datacenter. Users interact with F1 Query through its client library that sends requests to one of several dedicated servers we refer to hereafter as F1 servers. The F1 Master is a specialized node within the datacenter that is responsible for the run-time monitoring of query execution and maintenance of all F1 servers at that datacenter. Small queries and transactions begin executing on the immediate F1 server that receives the requests. F1 schedules larger queries for distributed execution by dynamically provisioning execution threads on workers from a worker pool. The largest queries are scheduled for execution in a reliable batch execution mode that uses the MapReduce framework. Final results are collected on the F1 server and then returned to the client. F1 servers and workers are generally stateless, allowing a client to communicate with an arbitrary F1 server each time. Since F1 servers and workers do not store data, adding new F1 servers or workers does not trigger any data redistribution cost. Therefore an F1 Query deployment at a datacenter can easily scale out by adding more servers or workers.

Query Execution. Users interact with F1 Query through its client library. A client's query request may arrive at one of many F1 servers. Upon arriving at an F1 server, the F1 server first parses and analyzes the SQL query, then extracts the list of all data sources and sinks that the query accesses. If any data sources or sinks are not available in the local datacenter, and there are F1 servers at other datacenters closer to the data sources or sinks, the F1 server sends the query back to the client with information about the optimal set of datacenters available for running the query. The client then resends the query to an F1 server at the target datacenter for execution. We note that while disaggregation of storage and compute and a high-performance network fabric have obviated many locality concerns within the datacenter, choosing a datacenter *close to* data from a set of many geographically distributed datacenters still has a large impact on query processing latency.

Query execution begins on the F1 server with a planning phase in which the optimizer converts the analyzed query abstract syntax tree into a DAG of relational algebra operators that are then optimized at both logical and physical levels. The final execution plan is then handed off to the execution layer. Based on a client-specified execution mode preference, F1 Query executes queries on F1 servers and workers in an *interactive* mode or in a *batch* mode (using the MapReduce framework) as shown in Figure 2.

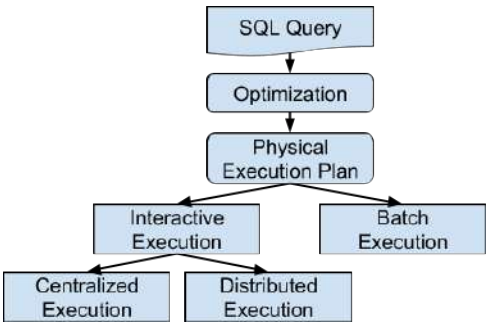

Figure 2: Query Execution Phases in F1

For interactive execution, the query optimizer applies heuristics to choose between single-node centralized execution and distributed execution. In centralized execution, the server analyzes, plans, and executes the query immediately at the first F1 server that receives it. In distributed mode, the first F1 server to receive the query acts only as the query coordinator. That server schedules work on separate workers that then together execute the query. The interactive execution modes provide good performance and resource-efficient execution for small and medium-sized queries.

Batch mode provides increased reliability for longer-running queries that process large volumes of data. The F1 server stores plans for queries running under batch mode in a separate execution repository. The batch mode distribution and scheduling logic asynchronously runs the query using the MapReduce framework. Query execution in this mode is tolerant to server restarts and failures.

Data Sources. F1 servers and workers in a datacenter can access data not just in the datacenter where they reside, but also in any other Google datacenter. The disaggregation of processing and storage layer enables data retrieval from a variety of sources, ranging from distributed storage systems like Spanner and Bigtable to ordinary files with varying structures such as comma-separated value text files (CSV), record-oriented binary formats, and compressed columnar file formats such as ColumnIO [51] and Capacitor [6]. F1 Query provides consistent and/or repeatable reads for data sources that support it, including data managed by the Spanner storage service.

To support queries over heterogeneous data sources, F1 Query abstracts away the details of each storage type. It makes all data appear as if it is stored in relational tables (with rich structured data types in the form of Protocol Buffers [9]; see Section 7.2) and enables joining data stored in different sources. It uses a global catalog service to maintain and retrieve meta-information about data sources stored in different formats and systems. F1 Query also allows querying sources not available through the global catalog service. In such cases, the client must provide a DEFINE TABLE statement that describes how to represent the underlying data source as a relational database table. Below, we show an example for retrieving data from a Colossus file in CSV format. F1 Query must know the location and type of the file as well as the names and types of the columns contained within. Note that different data sources may require different information to describe their structure depending on their unique properties.

```
DEFINE TABLE People(
   format = 'csv',
   path = '/path/to/peoplefile',
   columns = 'name:STRING,
              DateOfBirth:DATE');
SELECT Name, DateOfBirth FROM People
WHERE Name = 'John Doe';
```

While F1 Query natively supports the most widely-used data sources within Google, clients occasionally need to access data through a mechanism that is not known in advance. For this purpose F1 supports adding a new custom data source using an extension API called the Table-Valued Function (TVF) described in greater detail in Section 6.3.

Data Sinks. The output of queries can be returned to the client, but a query can also request that its output should be stored into an external data sink. Sinks may comprise files in various formats or otherwise use a variety of remote storage services. As with data sources, sinks may be either tables managed by the catalog service or manually specified targets. Managed tables are created by a CREATE TABLE statement. They are by default implemented as files stored on the Colossus file system. Manually specified storage targets are specified using the EXPORT DATA statement, using a specification that is similar to the corresponding DEFINE TABLE specification for reading back the same data. In addition to these options, queries can also create session-local temporary tables.

Query Language. F1 Query complies with the SQL 2011 standard, with extensions to support querying nested structured data. F1 Query supports standard SQL features including left/right/full outer joins, aggregation, table and expression subqueries, WITH clauses, and analytic window functions. For structured data, F1 Query supports variable length ARRAY types, as well as STRUCTs which are largely similar to SQL standard row types. Array types are well supported, including features like `UNNEST(array)` that pivots an array into a table with rows. F1 Query also provides support for Protocol Buffers [9], a format for exchange of structured data that used pervasively at Google. Section 7.2 covers this support in detail. [12] describes the shared SQL dialect used by F1 Query, Dremel [51]/BigQuery [3] and Spanner SQL [12], allowing users and applications to move between these systems with minimal overhead.

3. INTERACTIVE EXECUTION

By default, F1 Query executes queries in a synchronous online mode called *interactive* execution. F1 Query supports two types of interactive execution modes: central and distributed. During the planning phase, the optimizer analyzes the query and determines whether to execute it in central or distributed mode. In central mode, the current F1 server executes the query plan immediately using a single execution thread. In contrast, in distributed mode, the current F1 server acts as the query coordinator. It schedules work on other processes known as *F1 workers* that then together execute the query in parallel. In this section, we describe F1 Query interactive execution in detail.

3.1 Single Threaded Execution Kernel

Figure 3 depicts a SQL query and the resulting query plan for *central mode* execution. In this mode, F1 Query uses a single-threaded execution kernel. The rectangular boxes shown are operators within the execution plan. Single-threaded execution processes tuples in batches of 8 KiB using a recursive pull-based model. The

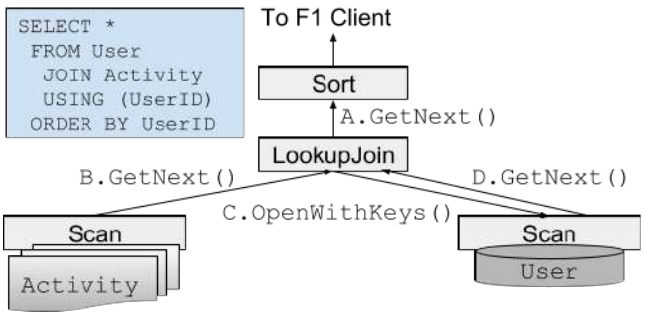

Figure 3: Central Query Execution on an F1 Server

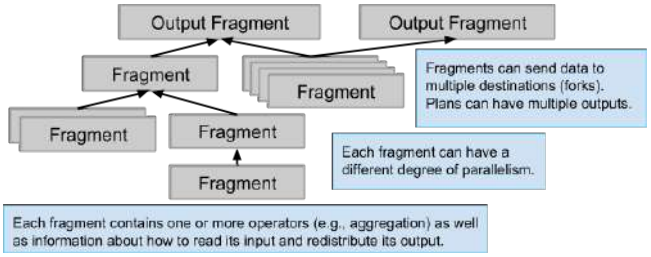

Figure 4: Fragments in Distributed Query Execution

execution operators recursively call `GetNext()` on the underlying operators until the leaf operators retrieve batches of tuples. The leaves are typically *Scan operators* that read from data sources. Each data source has its own scan operator implementation with feature sets depending on the type of data source. Some sources only allow full-table scans while others also support key-based index lookups. Some sources also support pushdown of simple filter expressions on non-key fields. A separate ARRAY scan operator produces rows from an array-typed expression as needed. For input data that may contain protocol buffers, all scan operators support protocol buffer decoding immediately at the data source scan node, ensuring the executor does not pass large encoded protocol buffer blobs around unless they are needed in their entirety. Instead, each scan operator immediately extracts the minimal set of fields the query requires (this is discussed in more detail in Section 7.2). F1 Query also supports several high performance columnar data sources that store the fields of protocol buffers or SQL structs separately and do not require any protocol buffer decoding at all.

F1 Query supports several join operators including lookup join (index nested-loop join), hash join, merge join, and array join. The hash join operator implements a multi-level recursive hybrid hash join with disk spilling to the Colossus distributed filesystem. The lookup join operator reads rows containing keys from its left input, and uses these keys to perform index lookups on its right input (which must be a scan operator). The merge join operator merges two inputs that share the same sort order. F1 Query also has an integrated scan/join operator for Spanner tables that implements a merge join on data streams from the underlying tables. An array join is a correlated join to an array scan where the array expression refers to the left input of the join, written in the SQL query as `T JOIN UNNEST(f(T))` for array-valued expression `f()`.

Besides scans and joins, F1 Query has operators for projection, aggregation (both sort-based and disk spilling), sorting, unioning, and analytic window functions. All execution operators, including scans and joins, include built-in support for applying filter predicates on their output rows and for LIMIT and OFFSET operations.

3.2 Distributed Execution

The optimizer generates a distributed execution plan when it detects that such a plan is best for the input tables to be scanned with high parallelism using partitioned reads. In this case, the query execution plan is split into query *fragments* as shown in Figure 4. Each fragment is scheduled on a group of F1 worker nodes. The fragments execute concurrently, with both pipelining and bushy parallelism. The worker nodes are multi-threaded and some workers may execute multiple independent parts of the same query.

The optimizer employs a bottom-up strategy to compute the plan fragment boundaries based on the input distribution requirements of each operator in the query plan. Each individual operator can have a requirement for distribution of its input data across the workers. If present, the requirement is usually *hashed on some set of fields*. Typical examples include grouping keys for aggregations or join keys for hash joins. When this requirement is compatible with the distribution of tuples from the input operator, the optimizer plans both operators inside the same fragment. Otherwise, it plans an exchange operator between two operators to generate a fragment boundary.

The next step is to select the number of parallel workers for each fragment (see Figure 4). Fragments operate with independent degrees of parallelism. The underlying data organization within table scans of leaf operators determines the initial parallelization, with an upper bound. A width calculator then recursively propagates this information up the query plan tree. For instance, a hash join between a 50-worker and a 100-worker fragment will be executed using 100 workers, to accommodate the larger of the two inputs.

The following query illustrates distributed execution:

```
SELECT Clicks.Region, COUNT(*) ClickCount
FROM Ads JOIN Clicks USING (AdId)
WHERE Ads.StartDate > '2018-05-14' AND
      Clicks.OS = 'Chrome OS'
GROUP BY Clicks.Region
ORDER BY ClickCount DESC;
```

This query involves two tables: `Ads` is a Spanner table for storing advertisement information, and `Clicks` is a table that stores ad clicks, defined in Mesa, one of Google's analytical data warehouses. This query finds all ad clicks that happened on *Chrome OS* with ad starting date after *2018-05-14*. It then aggregates the qualifying tuples to find the clicks per region, and sorts by descending number of clicks.

A possible plan for this query is shown in Figure 5. During execution, data streams bottom up through each of the operators until reaching the aggregation and sort operators. One thousand workers each scan data from the Clicks table. The query planner pushes down the filter `Clicks.OS = 'Chrome OS'` into the Mesa scan itself such that only rows satisfying the filter are returned from Mesa to F1 workers. Two hundred workers handle scanning of the `Ads` table with filter `Ads.StartDate > '2018-05-14'`. Data from both scans flows into a hash join operator and then the same F1 worker performs a partial aggregation over the join results. Finally, the F1 server performs the full aggregation and returns sorted output to the client.

3.3 Partitioning Strategy

In the distributed execution mode, F1 Query executes multiple fragments in parallel. The execution and data flow can be viewed as a DAG as shown in Figure 4. The data moves across each fragment boundary by being *repartitioned* using an exchange operator.

For each tuple, the sender applies a *partitioning function* to determine the destination partition for that tuple. Each partition number corresponds to a specific worker in the destination fragment.

The exchange operation is implemented using direct Remote Procedure Calls (RPCs, for short) from each source fragment partition to all destination fragment partitions, with flow control between each sender and receiver. This RPC-based communication mode scales well up to thousands of partitions per fragment. Queries requiring higher parallelism generally run in batch execution mode (described in Section 4). F1 Query's exchange operator runs locally within a datacenter, taking advantage of Google's Jupiter network [56]. Jupiter allows each server in a cluster of tens of thousands of hosts to communicate with any other server in the same cluster with sustained bandwidth of at least 10 Gb/s.

The query optimizer plans each scan operator as a leaf in the query execution plan along with a desired parallelism of N workers. To execute a scan in a parallelized way, the work must be distributed so that each scan worker produces a non-overlapping subset of the tuples, and all the workers together produce the complete output. The query scheduler then asks the scan operator to *partition itself* across N partitions. In response, the scan operator produces N or more partition descriptions. To achieve this, the scheduler then schedules copies of the plan to run on N workers, and sends each worker one of the partition descriptions obtained previously. Each worker then produces the subset of the data described by its partition description. In some cases, the actual number of partitions (for example, the number of data files for a file-based table) may exceed N, in which case the query executor dynamically assigns partitions to available workers over time. This approach avoids long tail latency for a scan arising from skew.

Some operators are executed in the same plan fragment as one of their inputs. For instance, lookup join is executed in the same fragment as its left input, processing lookups only for the tuples produced by the same partition of this input. In contrast, as shown in Figure 5, the execution of a hash join generally requires multiple fragments, each with multiple partitions. The query optimizer plans each input scan operator (or other subplan) in a separate fragment unless the input operator's data distribution is already compatible with the hash join keys. Each of these source fragments (SCAN `Clicks` and SCAN `Ads` in Figure 5) sends its data to the same destination fragment (shown on the right in Figure 5) that contains the hash join operator. Both input fragments send their data using the same partitioning function that is based on a hash of the join keys. This ensures that all rows with the same join keys end up in the same destination partition, allowing each hash join partition to execute the join for a particular subset of the key space.

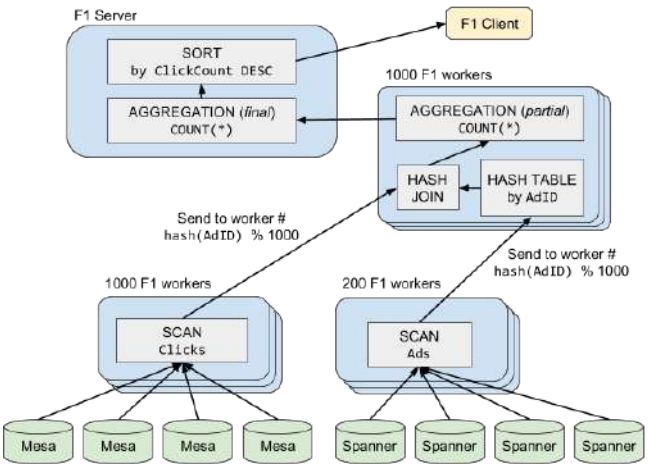

Figure 5: Distributed Query Execution Example

The aggregation operator also generally requires a repartitioning. For aggregation with grouping keys, the query plan repartitions the input tuples by a hash of the grouping keys, and sends these tuples to a destination fragment with the aggregation operator. For aggregation without grouping keys, all tuples are sent to a single destination. Figure 5 contains an example of aggregation without grouping keys. As can be seen in the figure, the aggregation is optimized by adding a second aggregation operator before the exchange operator which performs best-effort in-memory partial aggregation. This reduces the amount of data transferred, and for aggregation with grouping keys it mitigates the adverse impact of hot grouping keys during the full aggregation at the destination fragment.

As discussed earlier, the execution plans in F1 are DAG shaped, potentially with multiple roots. For forks in the data flow DAG, a plan fragment repartitions to multiple destination fragments, each with different partitioning functions. These DAG forks implement *run-once* semantics for SQL WITH clauses and identical subplans that are deduplicated by the optimizer. The DAG forks are also used for other complex plans e.g., analytic functions and multiple aggregations over DISTINCT inputs. DAG forks are sensitive to different data consumption speeds in consumer fragments, as well as to distributed deadlocks if multiple branches block when merging again later. Examples include self hash-joins from DAG forks that attempt to initially consume all tuples during the build phase. Exchange operators that implement DAG forks address these problems by buffering data in memory and then spilling data to Colossus when all consumers are blocked.

3.4 Performance Considerations

Primary causes for query performance issues in F1 Query include skew and sub-optimal data source access patterns. Hash join can be especially sensitive to hot keys from both inputs. Hot keys in the input that are loaded into the hash table (the *build input*) can lead to spilling, because one worker will need to store more tuples than others. Hot keys in the other input (the *probe input*) can generate CPU or network bottlenecks. For cases when one input is small enough to fit in memory, F1 Query supports a *broadcast* hash join that reads a small build input and broadcasts copies of all resulting tuples to all hash join workers. Each worker then builds an identical copy of the hash table. This broadcast hash join is not sensitive to skew, although it is sensitive to unexpectedly large build inputs.

During query execution, all lookup joins retrieve remote data using index keys. A naive key-by-key implementation would result in very slow execution due to the long tail in latency distribution of the underlying data sources, which are generally distributed systems themselves. For this reason F1 Query's lookup join operator uses large batches of outer rows. Such large batches allow for deduplication if the same lookup key is requested multiple times in the same batch. Scan operator implementations can also use the larger batches for optimizing data retrieval. For instance, partitioned data sources use the larger batch to find multiple keys that must be read from the same remote data source partition, merging them into a single efficient data source access. If the number of required remote requests for a batch exceeds the maximum number of parallel requests to the data source, tail latency from the underlying storage system is hidden, as requests can complete out of order, and longer-running requests do not block progress of other, shorter requests.

Skew and undesirable access patterns also frequently arise when placing lookup joins directly over their left inputs. Depending on the input data distribution, the data source access pattern can be ar-

bitrary, and there may be no deduplication at all because requests for the same keys are spread across the fragment partitions. Stacking multiple lookup joins results in skew when certain keys join with disproportionately more rows than others during the sequence. To counter these effects, the query optimizer has the capability to repartition the left input using one of several partitioning functions. The choice of partitioning function determines the data source access pattern of the lookup join, which has a strong impact on performance. For instance, hash partitioning ensures that each key originates from only one node, enabling deduplication of lookups, but for a data source the access patterns from each node will still look like random accesses. Range partitioned data sources such as Spanner and Bigtable benefit heavily from *key space locality* during lookups: when keys are concentrated in a small range of the key space, they are likely to reside on the same data source partition and can be returned to the F1 server as part of one single remote data source access. One way to exploit this is to use an explicit static range partitioning to assign a fixed key range to each destination fragment's partition, but this strategy is sometimes sensitive to skew. A better range-based strategy called *dynamic range repartitioning* computes an individual range partitioning function in each sender based on local distribution information. This is based on the principle that the distribution observed at one input plan fragment partition often closely approximates the overall data distribution. In many cases this results in a lookup pattern with much higher locality in the key space than other partitioning strategies. In addition, it produces a perfectly even workload distribution over the workers that perform the lookups. We have observed this strategy outperforms statically determined *ideal* range partitionings based on the key distribution in the lookup data source, in particular in cases when the left input is skewed and uses only a subset of the key space. Dynamic range repartitioning also adaptively responds to temporarily hot keys in the input data stream by spreading them out over more destination nodes as opposed to static range partitionings which create temporary hotspots.

F1 Query operators generally execute in memory without checkpointing to disk and stream data as much as possible. This avoids the cost of saving intermediate results to disk and lets queries run as fast as it is possible to consume input data. When combined with aggressive caching in data sources, this strategy enables complex distributed queries to run to completion in tens or hundreds of milliseconds [50]. In-memory execution is sensitive to F1 server and worker failures. The client library combats this by transparently retrying failed queries. In practice, queries that run for up to an hour are sufficiently reliable, but queries with longer runtimes may fail repeatedly. F1 Query's batch execution mode becomes a superior choice in these cases, as described in the next section.

4. BATCH MODE EXECUTION

F1 Query supports both interactive analysis as well as large-scale transformations over large amounts of data running for an extended time. These large-scale transformations typically process ETL (Extract-Transform-Load) workflows. Many of these ETL processing pipelines at Google were historically developed using MapReduce or FlumeJava [19], using mostly custom code for data transformation. Although customized ETL processing pipelines are effective, they incur a high development and maintenance cost. In addition, custom pipelines are not very amenable to many useful optimizations that a SQL query optimizer can perform, such as filter pushdown or attribute pruning. For instance, hand-written pipelines may needlessly pass large data structures between stages when only a small number of fields are needed, because the additional effort to optimize this is prohibitive and adds too much maintenance overhead. The declarative nature of SQL makes such manual optimizations unnecessary, and therefore it is preferable to use SQL for such pipelines.

The in-memory processing model of the interactive modes is not suited to handle worker failures, which are likely to occur during long-running queries. To address this challenge, F1 Query added a new mode of execution, called *batch mode*. Batch mode allows long-running queries to execute reliably even when there are F1 server or worker failures. In addition, it also handles *client* failures, by allowing clients to submit queries for asynchronous processing and then disconnecting.

Built on top of the F1 Query framework shown in Figure 2, F1 Query batch mode shares the same query specification, query optimization, and execution plan generation components with the two interactive modes. The key difference between the modes happens during execution scheduling. In the interactive modes, the query executes synchronously. The F1 server oversees the progress of the entire query until it completes. In contrast, for batch mode, the F1 server asynchronously schedules the query for execution. A central registry records progress of the query. This architecture imposes the following challenges:

- In batch mode, query plan execution operators must communicate differently since the query plan fragments execute asynchronously. In the distributed interactive mode, all fragments are active at the same time, and communicate using remote procedure calls. This not feasible in batch mode since different fragments of a query execute at different times.
- Since batch queries are long-running, we must account for the possibility of transient failures during execution, including machine restarts. This requires a fault-tolerant mechanism to persist intermediate states of the queries and guarantee forward progress.
- A new higher-level service framework is required to track the execution of thousands of batch queries at different execution stages to ensure that all are eventually completed.

In Section 4.1, we discuss in detail how F1 Query batch mode tackles the first two challenges, and then we cover the service framework in Section 4.2.

4.1 Batch Execution Framework

Batch mode uses the MapReduce (MR) framework as its execution platform. At an abstract level, each plan fragment (see Figure 4) in a query plan can be mapped to a MapReduce stage. Each stage in the processing pipeline stores its output to the Colossus file system. This communication model enables asynchronous execution of different MapReduce stages while providing the necessary fault-tolerance. When entire MapReduce stages fail, they can be restarted because their inputs are stored on Colossus. Failures during a MapReduce stage are tolerated due to the inherent fault-tolerance provided by the MapReduce framework.

In its most simplified form, one can map the plan fragments in the F1 Query execution plan to a MapReduce stage. However, F1 Query optimizes that in a way that is similar to the MSCR fusion optimization of FlumeJava [19]. In this optimization, leaf nodes are abstracted as a map operation, while internal nodes are be abstracted as a *reduce* operation. This mapping, however, results in a *map-reduce-reduce* type of processing, which does not completely correspond to the MapReduce framework. F1 Query batch mode resolves this problem by inserting a special map operator that is an identity function. This way, a *map-reduce-reduce* processing can be split into two MapReduce stages: *map-reduce* and *map<identity>-reduce*. Figure 6 illustrates this mapping from

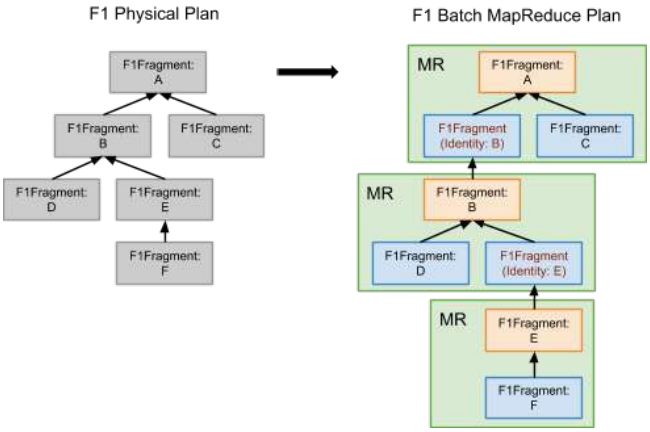

Figure 6: Mapping a Physical Plan to a Batch Mode MapReduce Plan

a regular physical execution plan to a batch mode MapReduce plan. As shown, the query plan on the left is mapped to only three MapReduce stages instead of the default mapping, which would have resulted in six MapReduce stages. A further improvement that we have not yet implemented would be to use a framework like Cloud Dataflow [10] that supports the *map-reduce-reduce* type of processing natively.

Between fragments, F1 Query's distributed interactive mode sends data over the network via RPCs. Batch mode, instead, materializes the data into staging files, reads it back, and feeds into the next plan fragment. This is achieved by a common I/O interface for the plan fragment executor, which is implemented by both modes. Furthermore, in distributed interactive mode, every node in the query execution plan is live simultaneously, allowing for parallelism through pipelining. In contrast, in batch mode there is no pipelining: MapReduce stages only start once all their inputs are completely available. Batch mode does support *bushy* parallelism, i.e., independent MR stages can execute in parallel.

Note that F1 Query batch mode operates at very large scale, and incurs a large data materialization overhead for every exchange operator in the query plan. As such, it is beneficial to reduce the number of exchange operators in the plan where possible, especially when dealing with very large tables. One method of avoiding exchange operators is by replacing a hash join with a lookup join. For joins where the smaller input is too large for a broadcast hash join, or where there is significant skew, batch mode can materialize the smaller input into disk-based lookup tables called sorted string tables (SSTables) [20]. It then uses a lookup join operator, in the same fragment as the larger input, to look up into these tables, thereby avoiding a costly repartitioning on the larger input. The lookups use a distributed caching layer to reduce disk I/O.

4.2 Batch Service Framework

The F1 Query batch mode service framework orchestrates the execution of all batch mode queries. It is responsible for registering incoming queries for execution, distributing queries across different datacenters, and scheduling and monitoring the associated MapReduce processing. Figure 7 shows the service framework architecture. When an F1 Client issues a query for running in batch mode, one of the F1 servers receives it. It then generates an execution plan, and registers the query in the *Query Registry*, which is a globally distributed Spanner database that tracks the metadata for all batch mode queries. The Query Distributor component of the service then assigns the query to a datacenter, choosing the data-

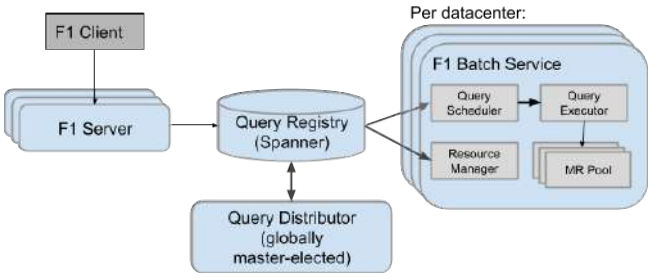

Figure 7: Batch Mode Service Framework

center based on load balancing considerations and the availability of data sources needed for execution.

The query is then picked up by the framework components that run in the target datacenter. Each datacenter has a Query Scheduler that periodically retrieves new assigned queries from the Query Registry. The scheduler creates a dependency graph of the query execution tasks, and when a task is ready to execute and resources are available, the scheduler sends the task to a Query Executor. The Query Executor then uses the MapReduce worker pool to execute the task.

The service framework is robust, with resilience features at every level. All of the components have redundancy, including the global Query Distributor which is replicated and master-elected, and the Query Scheduler which has multiple redundant instances per datacenter. All execution state of a query is maintained in the Query Registry, which allows all components to be effectively stateless and replaceable. Within a datacenter, failed MapReduce stages are retried several times. If a query stalls entirely, e.g. due to a datacenter outage, the Distributor reassigns the query to an alternate datacenter, which restarts execution from the beginning.

5. QUERY OPTIMIZER ARCHITECTURE

Query optimizer development is notoriously complex. F1 Query mitigates this by reusing the same logic for planning all queries regardless of execution mode. Even though interactive and batch execution modes use significantly different execution frameworks, both use the same plans and the same execution kernel. In this way all query planning features implemented in the F1 Query optimizer automatically work for both execution modes.

The high level structure of the F1 Query optimizer is shown in Figure 8, and draws inspiration from Cascades [35] style optimization. This infrastructure shares some design principles and terminology with Spark's Catalyst planner [11] because of early conversations on this topic between members of the F1 Query and Catalyst teams. The first step is to call Google's SQL resolver to parse and analyze the original input SQL and produce a resolved abstract syntax tree (AST). The optimizer then translates each such AST into a relational algebra plan. A number of rules execute on the relational algebra until reaching a fixed-point condition to produce a heuristically determined *optimal* relational algebra plan. The optimizer

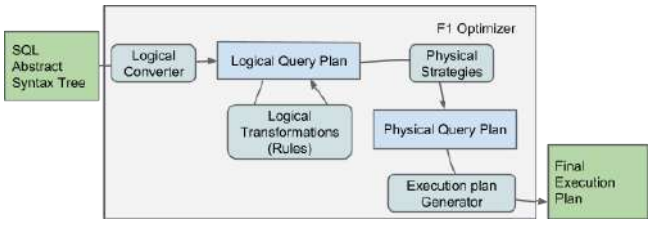

Figure 8: F1 Optimizer

then converts the final algebra plan into a physical plan including all data source access paths and execution algorithms. The optimizer completes its work by converting the physical plan into final data structures suitable for query execution, and passes them to the query coordinator for execution.

It should be noted that the F1 Query optimizer is based mainly on heuristic rules. It uses statistical data properties to some extent when present. However, due to diversity in data sources the typical F1 query only uses statistics from certain sources depending on what is feasible to collect in advance.

5.1 Optimizer Infrastructure

All stages of the optimizer are based on a common infrastructure layer for representing plan trees of various kinds, and transformations that operate on them. All plan tree structures are *immutable*: transformation stages must build new operators to change the query plan. This property enables exploratory planning as well as subtree reuse. To mitigate negative performance impact from multiple constructions and destructions of data structures, the optimizer constructs all data structures in a memory arena and destructs it outside the critical path of the query.

The optimizer has separate tree hierarchies for expressions, logical plans, and physical plans. The boilerplate code for the hundreds of tree node kinds is generated from only ~3K lines of Python code accompanied by ~5K lines of Jinja2 [7] templates, resulting in ~600K lines of C++. The generated code enables a *domain specific language* (DSL) for query planning and contains methods to compute a hash for each tree node, to perform tree equality comparisons, as well as other helpers suitable for storing trees in standard collections and representing them in testing frameworks. The use of code generation saves F1 Query engineers considerable time, reduces mistakes during development, and enables the effective rollout of new features across tree hierarchies.

All relational algebra rules and plan conversion stages inspect and manipulate trees using a C++ embedded DSL for tree pattern matching and building. Because of code generation and C++ templates, tree pattern expressions perform as well as optimized handwritten code. At the same time, they are more concise than hand written code and more clearly express the intent of each rewrite.

5.2 Logical Query Plan Optimization

When the SQL query analyzer receives the original query text, it produces a resolved abstract syntax tree (AST). The F1 Query optimizer then converts this AST into a relational algebra tree. It then applies logical rewrite rules to apply heuristic updates to improve the query plan. Rules are organized into batches and each batch runs either exactly once or until reaching a fixed point. Rules applied include filter pushdown, constant folding, attribute pruning, constraint propagation, outer join narrowing, sort elimination, common subplan deduplication, and materialized view rewrites.

Data sources in F1 Query may include structured protocol buffer data within relational table columns, and all rules have first-class knowledge of protocol buffers. For example, the core attribute pruning rule recursively pushes down extraction operation expressions for individual protocol buffer fields down the query plan as far as possible. If such extractions travel all the way to the leaves of the query plan, it often becomes possible to integrate them into scan operations to reduce the number of bytes read from the disk or transferred over the network.

5.3 Physical Query Plan Construction

Based on the relational algebra plan, the optimizer then creates a physical plan tree, which represents the actual execution algorithms and data source access paths. Physical plan construction logic is encapsulated in modules called *strategies*. Each strategy attempts to match against one specific combination of relational algebra operators. The strategy then produces physical operators to implement the matched logical operators. For example, one strategy handles lookup joins only, detecting logical joins of tables with suitable indexes and then producing physical lookup join operators between them. Each resulting physical operator is represented as a class that tracks multiple data properties, including distribution, ordering, uniqueness, estimated cardinality, and volatility (among others). The optimizer uses these properties to determine when to insert an *exchange operator* to repartition input tuples to a new data distribution required by the next operator. The optimizer also uses the physical plan properties to decide whether to run a query in central or distributed mode. When any scan is deemed to be too expensive for a central query, e.g. because it is a full table scan, then the entire query is planned as a distributed query.

5.4 Execution Plan Fragment Generator

The final stage of the query optimizer converts the physical query plan into a series of plan fragments suitable for direct execution. This execution plan fragment generator converts physical plan tree nodes into corresponding execution operators with a fragment boundary at each exchange operator. The generator also takes responsibility for computing a final degree of parallelism for each fragment, starting at leaf fragments containing distributed table scans and propagating upwards to the root of the query plan.

6. EXTENSIBILITY

F1 Query is extensible in various ways: it supports custom data sources as well as user defined scalar functions (UDFs), aggregation functions (UDAs), and table-valued functions (TVFs). User defined functions can use any type of data as input and output, including Protocol Buffers. Clients may express user-defined logic in SQL syntax, providing them with a simple way of abstracting common concepts from their queries and making them more readable and maintainable. They may also use Lua [42] scripts to define additional functions for ad-hoc queries and analysis. For compiled and managed languages like C++ and Java, F1 Query integrates with specialized helper processes known as UDF servers to help clients reuse common business logic between SQL queries and other systems.

UDF servers are RPC services owned and deployed separately by F1 Query clients. They are typically written in C++, Java, or Go, and execute in the same datacenters as the F1 servers and workers that call them. Each client maintains complete control over their own UDF server release cycle and resource provisioning. The UDF servers expose a common RPC interface that enables the F1 server to find out the details of the functions they export and to actually execute these functions. To make use of the extensions provided by a UDF server, the F1 Query client must provide the address of the UDF server pool in the query RPC that it sends to the F1 server. Alternatively, owners of F1 databases may configure default UDF servers that will be made available to all queries that run in the context of that database. Even though F1 will communicate with UDF servers during query execution, they remain separate processes and isolate the core F1 system from failures in the custom functions.

SQL and Lua scripted functions do not use UDF servers, and there is no single central repository for their definitions. Instead, clients must always supply their definitions as part of the RPC that they send to F1 Query. Client tools, such as the F1 Query command line interface, gather function definitions from configuration files

and other sources loaded explicitly, and also from immediate commands. They subsequently pass all relevant function definitions as part of every RPC sent to F1 Query. F1 Query does provide a mechanism to group multiple SQL UDFs, UDAs and TVFs into *modules*. Client teams use modules to structure their custom business logic, improving maintainability and promoting reuse. Modules are presented to F1 Query in the same way as individual UDFs, through the query RPC that is sent to F1 Query.

6.1 Scalar Functions

F1 Query supports scalar UDFs written in SQL, Lua, and as compiled code through UDF servers. SQL UDFs allow users to encapsulate complex expressions as reusable libraries. They are expanded in-place where they are used in the query. For scripting languages like Lua, the query executor maintains a sandboxed interpreter to evaluate scripted functions at runtime. For example, the Lua UDF shown below converts a date value encoded as a string into an unsigned integer representing the corresponding Unix time:

```
local function string2unixtime(value)
  local y,m,d = match("(%d+)%-(%d+)%-(%d+)")
  return os.time({year=y, month=m, day=d})
end
```

Functions exported by UDF servers can be evaluated only within the projection execution operator. When parsing each query, the system generates a function expression for each UDF. The optimizer then moves all such expressions into projections. During execution, the projection operator buffers input rows and calculates their associated UDF argument values, up to a size limit. The worker then dispatches an RPC to the associated UDF server. UDF server latency is hidden by pipelining multiple RPCs. This allows for fairly high latency UDF implementations without impacting query latency.

6.2 Aggregate Functions

F1 Query also supports user-defined aggregate functions, which combine multiple input rows from a group into a single result. As with scalar functions, users can define UDAs in SQL and the query optimizer expands the definition at each call site. For compiled and managed languages, the system also supports hosting UDAs in UDF servers. A UDF server-based UDA definition must implement the typical UDA processing operations *Initialize, Accumulate, and Finalize* [31,43,44]. In addition, it must implement the *Reaccumulate* operation that is used to combine multiple aggregation buffers from partial aggregation (see Figure 5).

During execution, the aggregation operator processes input rows and buffers aggregation input for each UDA aggregate value in memory. When the sum of memory usage from all such buffered inputs in the hash table exceeds a certain size, the executor sends the existing aggregate values and the new inputs for each group over to the UDF server. The UDF server then calls the appropriate UDA operations to produce a new aggregate value for each group. The UDF servers are stateless, letting each F1 server distribute requests to many UDF server processes in parallel.

6.3 Table-Valued Functions

Finally, F1 Query exposes table-valued functions (TVF), a framework for clients to build their own user-defined database execution operators. Table-valued functions serve a variety of purposes to help extend the power of F1 Query. Salient examples include integrating machine-learning steps like model training during SQL query execution, which lets users consume data and then run advanced predictions in a single step. Development teams throughout the company can also add new TVF data sources as needed without any requirement to interact with core F1 Query developers or to restart running database servers.

A TVF can accept entire tables as well as *constant* scalar values as input, and uses those inputs to return a new table as output. A query can call the TVF by invoking it in the FROM clause, passing in scalar parameters, tables from the database, or table subqueries. For instance, this calls a TVF with scalar parameter and a database table to calculate advertising click activity for the past 3 days:

```
SELECT * FROM EventsFromPastDays(
    3, TABLE Clicks);
```

As with UDFs and UDAs, it is possible to define a TVF using SQL. Such TVFs are similar to parameterized views, where the parameters can be entire tables. They are expanded into the query plan before query optimization, so that the optimizer can fully optimize the TVF. The UDF called above might be defined using SQL as follows:

```
CREATE TABLE FUNCTION EventsFromPastDays(
    num_days INT64, events ANY TABLE) AS
    SELECT * FROM events
    WHERE date >= DATE_SUB(
        CURRENT_DATE(),
        INTERVAL num_days DAY);
```

Note that this example uses ANY TABLE to specify that the function can accept any table as an argument. In this situation, the TVF dynamically computes an output schema based on the actual input table of each query at analysis time, after which point the output schema remains fixed for the duration of that query's execution. It is also possible to specify that input tables must have a specific schema, in which case F1 Query enforces this invariant at query analysis time.

More complicated TVFs can be defined using UDF servers. A UDF server exposes a TVF definition using a function signature. This signature may include generic parameters like in the SQL TVF example. The TVF definition also provides a function to compute the output table schema for a specific call. Interestingly, this output schema may depend not only on the input table column types but also on the values of scalar constant arguments. Hence, the TVF uses this function to compute the output schema even if the signature contains no generic parameters. The TVF definition also exposes execution properties for the optimizer, such as whether a TVF on an input table can be parallelized by partitioning the input table by certain keys and then calling the TVF implementation on each partition separately.

The query optimizer chooses one of two operators to evaluate remotely-hosted TVFs over the network. The first operator applies specifically when the TVF contains no input table arguments. In this case, it represents a *remote data source*, and it is planned and executed like other data sources, including support for partitioned scans and lookup joins. Functions with input table arguments are handled by a specialized TVF execution operator. For both types of TVFs, the optimizer may push down filters, limits, and aggregation steps into the TVF itself, which may use them to reduce work.

The RPC protocol for remote TVF evaluation uses a persistent bidirectional streaming network connection to send input rows to the UDF server and receive output rows back, as shown in Figure 9. For remote data sources, the optimizer also sends an RPC call to the

UDF server to retrieve partition descriptions for the TVF so that multiple workers can scan the data source in parallel.

7. ADVANCED FUNCTIONALITY

7.1 Robust Performance

F1 Query identifies robustness of performance as a crucial issue in database query processing, and an important third dimension affecting user experience beyond efficiency and scalability. Robustness requires that performance gracefully degrades in the presence of unexpected input sizes, unexpected selectivities, and other factors. Without graceful degradation, users may see a *performance cliff*, i.e., a discontinuity in the cost function of an algorithm or plan. For example, the transition from an in-memory quicksort to an external merge sort can increase end-to-end sorting runtime by a factor of two or more once the entire input begins spilling into temporary files. Figure 10 shows an example of this discontinuity in the performance of the F1 Query sort operation with a cliff (measured before) and with the cliff removed (measured after). Cliffs create several problems, including unpredictable performance and a poor experience for the user; optimizer choices become error-prone because a small cardinality estimation error may be amplified into a large cost calculation error; and in parallel query execution, small load imbalances between compute nodes may turn into large disparities in elapsed runtimes.

F1 Query employs robust algorithms to prevent performance cliffs. The principal idea is that instead of using a binary switch at optimization time or at execution time, the execution operator incrementally transitions between modes of operation. For example, its sort operator spills only as much data from its in-memory workspace as required to make room for additional input in memory. Another example in sorting occurs during the transition to multiple merge steps, where one additional input byte could force all input records to go through two merge steps instead of only one [36]. F1 Query eliminates both of these cliffs from its implementation of sorting and aggregation. Successful examples of cliff avoidance or removal include SmoothScan [16] and dynamic destaging in hash joins [52]. Dynamic re-optimization would introduce a huge cliff if a single row "too many" will stop execution and re-start the compile-time optimizer.

7.2 Nested data in Google Protocol Buffers

Within Google, Protocol Buffers [9] are ubiquitous as a data interchange and storage format. Protocol Buffers are a structured data format with record types called *messages* and support for array-valued or *repeated* fields. Protocol Buffers have both a human-readable text format and a compact, efficient binary representation. They are a first-class data type in the F1 Query data model, and its SQL dialect has extensions for querying and manipulating individual messages, e.g. `msg.field` for field access, and `NEW Point(3 AS x, 5 AS y)` to create a new message. F1 Query also supports correlated subquery expressions and joins over repeated fields.

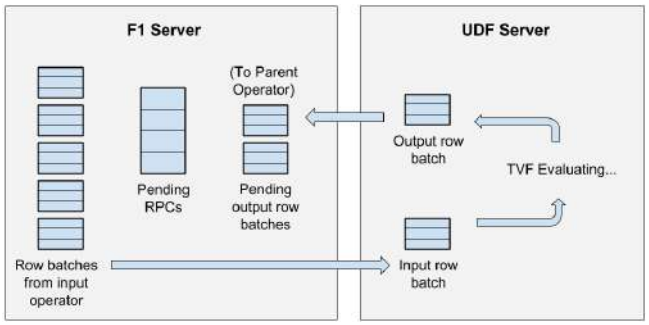

Figure 9: Remote TVF Evaluation

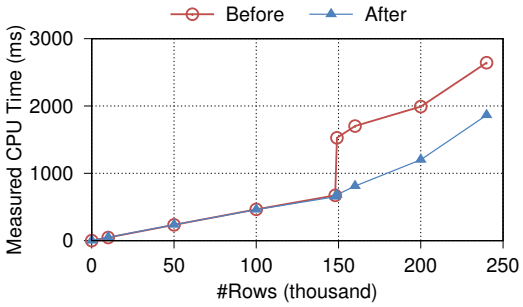

Figure 10: Cliff from Internal to External Sort

Querying protocol buffers presents many of the same challenges as semi-structured data formats like XML [18] and JSON [21], for which there is a rich body of research. Some key differences exist, however. Where JSON is entirely dynamically typed and often stored in human readable format, protocol buffers are statically typed and typically stored in a compact binary format, enabling much more efficient decoding. The binary encoding of protocol buffers is somewhat similar to the binary encoding of JSON objects used in MongoDB [2], but it is more efficient because fields are statically typed and identified by integers instead of strings. In addition, some data sources vertically decompose the messages into a columnar format [51], in a way similar to vertical shredding of documents in XML databases [29].

The exact structure and types of all protos referenced in a query are known at query planning time, and the optimizer prunes away all unused fields from data source scans. Within columnar data sources, this reduces I/O and enables efficient column-wise evaluation of filters. For record-oriented data sources that uses the row-wise binary format, F1 Query uses an efficient streaming decoder that makes a single pass over the encoded data and extracts only the necessary fields, skipping over irrelevant data. This is enabled only by the fixed definition of each protocol buffer type, and the integer field identifiers that are fast to identify and skip over.

8. PRODUCTION METRICS

In this section, we report query processing performance and volume metrics for a representative subset of the F1 Query production deployment. F1 Query is highly decentralized and replicated over multiple datacenters, using hundreds to thousands of machines at each datacenter. Although we do not report the proprietary details of our deployment, the reported metrics in this section are comprehensive and demonstrate the highly distributed, large scale nature of the system. We show the metrics over multiple days to demonstrate both their variability and stability. We also show our QPS growth metric and query latencies over multiple quarters, to illustrate how the system scales to support increasing demand without

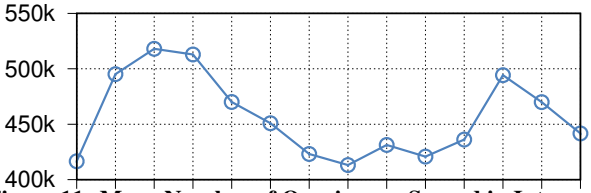

Figure 11: Mean Number of Queries per Second in Interactive Modes for Each Day of a Two-Week Period

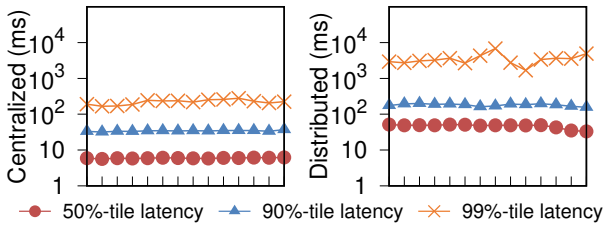

Figure 12: Latency of Interactive Queries (Centralized & Distributed Execution, y-axes are log scale)

performance degradation. F1 Query blurs the lines between workloads, so it is not feasible to report which fraction of traffic represents OLTP, OLAP, or ETL. However, the latency metrics reported across the execution modes show how F1 Query scales to process queries at every scale from tiny to huge, regardless of query intent.

F1 Query is used within Google by over 10,000 users each week. These users include both individuals performing ad hoc analysis and system users that represent the activity of entire products. F1 Query also serves as the SQL layer for 100s of production F1/Spanner databases. Figures 11 and 12 report the aggregate throughput and latency metrics of the F1 Query interactive execution subsystem. As shown in Figure 11, the mean throughput of the interactive subsystem over multiple days is around 450,000 queries per second which amounts to approximately 40 billion queries daily. We have observed that the system can easily handle peak throughput of up to 2X of the mean throughput without adversely impacting query latency. Figure 12 reports the latency numbers for both centralized and distributed interactive query execution. The 50th percentile, 90th percentile, and 99th percentile latencies for centralized queries are under 10 ms, under 50 ms, and under 300 ms, respectively. The latency numbers for the distributed execution are higher: 50 ms, 200 ms, and 1000+ ms. As the figure depicts, the 99th percentile for distributed execution exhibits much higher variability due to the large variance of long-running ad-hoc queries.

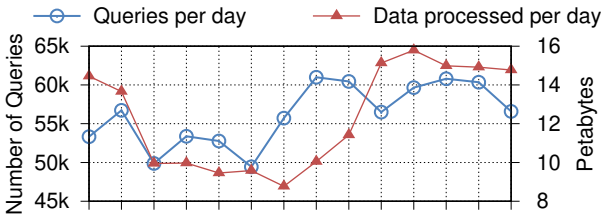

Figure 13: Batch Mode Daily Metrics

Figure 13 reports the number of queries executed in batch mode and the volume of data processed. The mean query throughput for batch mode execution is around 55,000 queries per day which is considerably smaller than the throughput of interactive execution. The main reason for the demand for batch mode query execution is much lower in that batch mode is only needed for running very large analysis queries and for running ETL pipelines. The mean query latency in batch mode is well below 500 seconds (under 10 minutes) and the maximum latency is as high as 10,000+ seconds

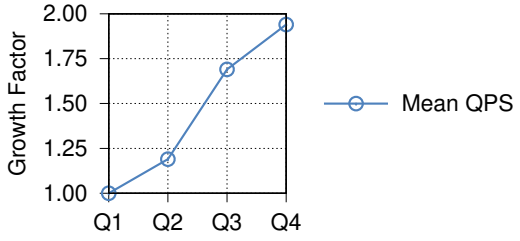

Figure 14: Long-term QPS Growth

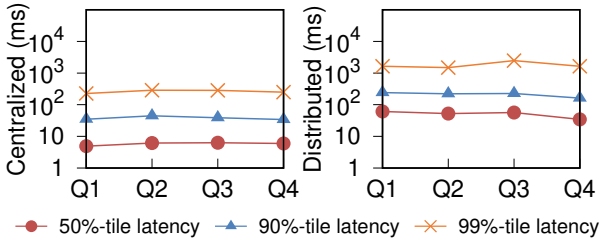

Figure 15: Latency of Interactive Queries over Multiple Quarters (y-axes are log scale)

(multiple hours). The query latency is higher in batch mode than interactive mode due to the complexity and the volume of data processed by such queries. Batch mode queries process about 8 to 16 petabytes of total input data per day. These numbers clearly establish the scale at which F1 Query operates within Google.

We conclude this section with a summary of long term query throughput growth in F1 Query over multiple quarters. As shown in Figure 14 in relative terms, the query throughput almost doubled in four quarters, and the scalable design of F1 Query allows it to handle this increased throughput without adversely impacting query latency as evident in Figure 15.

9. RELATED WORK

Distributed query processing has a long history in the database research literature [14,30,47,49,61]. In the late seventies and early eighties, the advent of computer networks allowed researchers to envision a database architecture distributed over multiple machines. Notable efforts include SDD-1 (A System for Distributed Databases) [14] initiated at the Computer Corporation of America and System R* [49] initiated at IBM Research in San Jose. While both projects resulted in significant advances in the areas of distributed transaction management, concurrency control, recovery, query processing, and query optimization, the final outcome from both projects was that distributed databases were not feasible primarily due to bandwidth and latency constraints in the network. In the same vein, there exists a large body of work in the area of parallel query processing in relational databases [15,27,28,32,34,41,45] in the context of multiprocessor and data-parallel architectures with multiple physical disk storage. Unlike distributed query processing, parallel databases have been highly successful. Much of the research carried out in the context of parallel databases has been commercialized by large DBMS vendors, especially for building large data warehouses for processing analytical queries.

Much of the early work as well as current efforts in the research arena have been primarily in the context of classical system architectures: client-server architectures as well as cluster-based architectures. As far as we are aware, relatively few comprehensive efforts exist to design a distributed and parallel database that leverages the computing and storage resources at the datacenter scale. Commercial DBMS vendors are starting to offer DBMS solutions [8, 13, 17, 37, 59] that are cloud-enabled with support for partitioning over multiple machines in the datacenter and processing queries against these partitions. Other parallel databases like Snowflake [24] also separate storage from computation similar to the design principles that originally shaped F1 [55]. However, these systems are all still tightly-coupled, even if they physically decouple storage from computation. They expect that data is natively stored in their own storage layer that they fully control, or that they can ingest data before processing it. On the other hand, F1 Query can be used to process queries against datasets stored in any format in any datacenter, with no assumptions on formats or need for extraction steps. Also, these systems often focus on one use case,

most commonly analytical query processing, whereas F1 Query covers all use cases. For instance, Impala [46] claims to be inspired by Google F1, and decouples storage from computation like F1 Query, but it was also created as an analytics query engine from the ground up, and supports no OLTP-style queries with associated constructs like F1 Query's lookup join.

Similar to F1 Query batch execution mode, systems like Pig Latin [53], Hive [58], and, until recently, Spark [62] implement query processing systems using a batch processing paradigm similar to MapReduce. Such systems provide mostly-declarative abstractions, with extensibility using custom code. The main benefit that F1 Query has over batch oriented systems lies in its versatility: in F1 Query, the same SQL queries can be applied to smaller amounts of data using the interactive execution modes, or applied to a huge amount of data and run in a reliable batch oriented mode. Some of these systems require using a custom data flow language and do not support SQL, which means that queries written for these systems also cannot easily be run on other systems, for instance on low-latency oriented systems.

Relational database management systems have long supported user-defined functions that consume and return values within SQL query execution [48, 60]. SQL/MapReduce [31] explored the idea of running TVFs in a separate process, running co-located with the database worker. F1 Query expands on this by disaggregating its UDF servers from the database entirely, allowing them to be deployed and scaled independently, and to be shared by F1 servers, workers, and batch mode MapReduce workers. Optimization support for opaque TVFs has also been explored in [54]. AWS Lambda [1] and Google Cloud Functions [5] use serverless architectures [40] to implement user-defined functions written in interpreted and managed languages only. F1 Query also supports the use of fully compiled languages like C++ and Go in its UDF servers to help run interactive queries with low latency, but in principle UDF servers could be implemented with a serverless architecture as well.

9.1 Related Google Technologies

F1 Query has some features in common with externally available Google systems Spanner and BigQuery. One critical similarity of all three systems is their shared SQL dialect, helping developers and analysts move between the systems with minimal overhead.

Spanner SQL [12] and F1 Query share many aspects, but they are different in one important area. The former is a single-focus SQL system operating on a transactional core whereas F1 Query is loosely coupled to its data sources. Spanner SQL also offers very fine grained query restartability whereas F1 Query's restartability is more coarse-grained.

BigQuery is Google's cloud data warehouse. Queries are served by Dremel [51], a widely used query engine for ad-hoc analysis within Google. It is optimized for analytic queries on columnar data and is capable of large-scale aggregations and joins. F1 Query supports these use cases and provides additional support for OLTP-style queries on data sources that support key lookups.

PowerDrill [39] is a query engine used at Google for interactive data analysis and exploration. It is a tightly coupled system that pre-processes data for a particular class of queries. In comparison, F1 Query has a much broader scope, but does not yet have an equivalent level of optimization for the data exploration use case.

The Tenzing system [22] used the MapReduce framework to execute SQL queries. Queries served by Tenzing have been migrated to either Dremel or F1 Query, with long running ETL style queries primarily served by F1 Query batch mode today. Similar to Tenzing, F1 Query batch mode uses MapReduce as its execution framework. Compared to the Tenzing service, the F1 Query batch mode service provides better fault tolerance, user isolation, and scheduling fairness. As a result, it provides higher throughput, and its query latency under high load is roughly 45% of Tenzing's.

FlumeJava [19] and Cloud Dataflow [4, 10] are modern replacements for MapReduce that allow pipeline operations to be specified at a higher level of abstraction, similar to e.g. Pig Latin. As batch oriented systems, they do not support interactive query execution. They do not natively support SQL. They do support some optimization of the data flow, though and they lack the ability to do certain optimizations like attribute pruning. Work is presently underway to extend F1 Query batch mode to take advantage of FlumeJava's improvements over classic MapReduce.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated that it is possible to build a query processing system that covers a significant number of data processing and analysis use cases on data that is stored in any data source. By combining support for all of these use cases in a single system, F1 Query achieves significant synergy benefits compared to the situation where many separate systems exist for different use cases. There is no duplicated development effort for features that would be common in separate systems like query parsing, analysis, and optimization, ensuring improvements that benefit one use case automatically benefit others. Most importantly, having a single system provides clients with a one-stop shop for their data querying needs and removes the discontinuities or "cliffs" that occur when clients hit the boundaries of the supported use cases of more specialized systems. We believe that it is the wide applicability of F1 Query that lays at the foundation of the large user base that the product has built within Google.

F1 Query continues to undergo active development to address new use cases and to close performance gaps with purpose-built systems. For instance, F1 Query does not yet match the performance of vectorized, columnar execution engines (e.g. Vectorwise [63]) because of its row-oriented execution kernel. A transition to a vectorized execution kernel is future work. F1 Query also does not support local caches for data in the query engine's native format, such as one naturally finds in shared-nothing architectures, since all data sources are disaggregated and remote. Currently, F1 Query relies on existing caches in the data sources, or remote caching layers such as TableCache [50]. To support in-memory or nearly-in-memory analytics, such as offered by PowerDrill [39], F1 Query would need to support local caching on individual workers and locality-aware work scheduling that directs work to servers where data is likely to be cached. The use of remote data sources also makes it harder to collect statistics for use in query optimization, but we are working to make them available so that F1 Query can use cost based optimization rules. And while F1 Query has excellent support for scaling out, we are working on techniques to improve how F1 scales *in*, for example, by running medium-sized distributed queries on only a few servers, thereby reducing the cost and latency of exchange operations.

Acknowledgements

We would like to thank Alok Kumar, Andrew Fikes, Chad Whipkey, David Menestrina, Eric Rollins, Grzegorz Czajkowski, Haifeng Jiang, James Balfour, Jeff Naughton, Jordan Tigani, Sam McVeety, Stephan Ellner, and Stratis Viglas for their work on F1 Query or feedback on this paper. We also thank interns and postdocs Michael Armbrust, Mina Farid, Liam Morris, and Lia Guy for their work on F1 Query. Finally, thank you to to the F1 SRE team for amazing F1 Query production support and help in scaling the service to 1000s of users.

11. REFERENCES

[1] AWS Lambda. https://aws.amazon.com/lambda/.

[2] BSON (binary JSON). http://bsonspec.org.

[3] Google BigQuery. https://cloud.google.com/bigquery.

[4] Google Cloud Dataflow. https://cloud.google.com/dataflow.

[5] Google Cloud Functions. https://cloud.google.com/functions/docs/.

[6] Inside Capacitor, BigQuery's next-generation columnar storage format. https://cloud.google.com/blog/big-data/2016/04/inside-capacitor-bigquerys-next-generation-columnar-storage-format.

[7] Jinja. http://jinja.pocoo.org.

[8] Oracle database cloud service. https://cloud.oracle.com/database.

[9] Protocol Buffers. https://developers.google.com/protocol-buffers.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[12] 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*, pages 331–343, 2017.

[13] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[14] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (SDD-1). *TODS*, 6(4):602–625, 1981.

[15] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the execution of relational database operations. *TODS*, 8(3):324–353, 1983.

[16] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth scan: Statistics-oblivious access paths. In *ICDE*, pages 315–326, 2015.

[17] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *SIGMOD*, pages 1021–1024, 2010.

[18] D. Chamberlin. Xquery: A query language for XML. In *SIGMOD*, pages 682–682, 2003.

[19] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[20] 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. *TOCS*, 26(2):4:1–4:26, 2008.

[21] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *WebDB*, pages 1–6, 2013.

[22] 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):1318–1327, 2011.

[23] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[24] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake elastic data warehouse. In *SIGMOD*, pages 215–226, 2016.

[25] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[26] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *CACM*, 53(1):72–77, 2010.

[27] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *ACM SIGMOD Record*, 19(4):104–112, 1990.

[28] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[29] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In *VLDB*, pages 1297–1300, 2004.

[30] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.

[31] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[32] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine GRACE. In *VLDB*, pages 209–219, 1986.

[33] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[34] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[35] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

[36] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3), 2006.

[37] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[38] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[39] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[40] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *HotCloud*, pages 33–39, 2016.

[41] W. Hong. *Parallel query processing using shared memory multiprocessors and disk arrays*. PhD thesis, University of California, Berkeley, 1992.

[42] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[43] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *SIGMOD*, pages 379–389, 1998.

[44] M. Jaedicke and B. Mitschang. User-defined table operators: Enhancing extensibility for ORDBMS. In *VLDB*, pages 494–505, 1999.

[45] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Relational algebra machine GRACE. In *RIMS Symposia on Software Science and Engineering*, pages 191–214. Springer, Berlin, Heidelberg, 1983.

[46] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[47] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[48] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *VLDB*, pages 294–305, 1988.

[49] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, pages 31–47. Springer, Berlin, Heidelberg, 1985.

[50] G. N. B. Manoharan, S. Ellner, K. Schnaitter, S. Chegu, A. Estrella-Balderrama, S. Gudmundson, A. Gupta, B. Handy, B. Samwel, C. Whipkey, L. Aharkava, H. Apte, N. Gangahar, J. Xu, S. Venkataraman, D. Agrawal, and J. D. Ullman. Shasta: Interactive reporting at scale. In *SIGMOD*, pages 1393–1404, 2016.

[51] 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):330–339, 2010.

[52] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, pages 468–478, 1988.

[53] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[54] A. Pandit, D. Kondo, D. E. Simmen, A. Norwood, and T. Bai. Accelerating big data analytics with collaborative planning in Teradata Aster 6. In *ICDE*, pages 1304–1315, 2015.

[55] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[56] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. *CACM*, 59(9):88–97, 2016.

[57] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[58] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[59] 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*, pages 1041–1052, 2017.

[60] H. Wang and C. Zaniolo. User defined aggregates in object-relational systems. In *ICDE*, pages 135–144, 2000.

[61] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys (CSUR)*, 16(4):399–433, 1984.

[62] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *CACM*, 59(11):56–65, 2016.

[63] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350, 2012.