Shasta: Interactive Reporting At Scale

Gokul Nath Babu Manoharan[α], Stephan Ellner[α], Karl Schnaitter[α], Sridatta Chegu[α], Alejandro Estrella-Balderrama[α], Stephan Gudmundson[α], Apurv Gupta[α], Ben Handy[α], Bart Samwel[α], Chad Whipkey[α], Larysa Aharkava[α], Himani Apte[α], Nitin Gangahar[α], Jun Xu[α], Shivakumar Venkataraman[α], Divyakant Agrawal[α], Jeffrey D. Ullman[β*]

[α]Google, Inc. [β]Stanford University

ABSTRACT

We describe Shasta, a middleware system built at Google to support interactive reporting in complex user-facing applications related to Google's Internet advertising business. Shasta targets applications with challenging requirements: First, user query latencies must be low. Second, underlying transactional data stores have complex "read-unfriendly" schemas, placing significant transformation logic between stored data and the read-only views that Shasta exposes to its clients. This transformation logic must be expressed in a way that scales to large and agile engineering teams. Finally, Shasta targets applications with strong data freshness requirements, making it challenging to precompute query results using common techniques such as ETL pipelines or materialized views. Instead, online queries must go all the way from primary storage to user-facing views, resulting in complex queries joining 50 or more tables.

Designed as a layer on top of Google's F1 RDBMS and Mesa data warehouse, Shasta combines language and system techniques to meet these requirements. To help with expressing complex view specifications, we developed a query language called RVL, with support for modularized view templates that can be dynamically compiled into SQL. To execute these SQL queries with low latency at scale, we leveraged and extended F1's distributed query engine with facilities such as safe execution of C++ and Java UDFs. To reduce latency and increase read parallelism, we extended F1 storage with a distributed read-only in-memory cache. The system we describe is in production at Google, powering critical applications used by advertisers and internal sales teams. Shasta has significantly improved system scalability and software engineering efficiency compared to the middleware solutions it replaced.

1. INTRODUCTION

Many business applications at Google aim to provide interactive data-rich UIs, allowing users such as advertisers, publishers and video content creators to manage and analyze their data. It is not uncommon for such applications to combine interactive reporting (OLAP) and update (OLTP) functionality in the same UI. For a representative example, consider AdWords, Google's search advertising product: In a Web UI, advertisers configure their campaigns. Campaign configurations are stored in the F1 database [13] in hundreds of normalized tables backed by Spanner storage [5]. Google serves ads based on these configurations, and log aggregation systems [1] use ad server logs to continuously update hundreds of metrics tables stored in Mesa [8], tracking the performance of the running ad campaigns in near real-time. Deeply integrating OLTP and OLAP functionality, the AdWords Web UI allows advertisers to view interactive reports with campaign performance metrics and to make changes to campaign configurations. Corresponding SOAP developer APIs allow advertisers to interact with their campaign and reporting data programmatically.

A challenge in scaling such applications in practice are vast database schemas with hundreds of tables, placing significant transformation logic between stored data and user-facing concepts. This "concept gap" tends to be particularly wide for OLAP functionality, posing non-trivial questions for application designers:

- **How to encapsulate complex data transformations?** Without proper abstractions, business logic tends to leak into inappropriate parts of the system or get duplicated, creating problems for code ownership, readability, consistency, and release velocity.

- **How to express the transformations?** Even if business logic is encapsulated well, the right paradigm to express it is non-obvious. Procedural languages such as Java or C++ are powerful, but can be cumbersome for expressing complex query operations on top of vast database schemas, e.g., joins of 50 tables. SQL on the other hand naturally captures relational operations, but doesn't scale gracefully to code artifacts of thousands of lines of code. Furthermore, applications that allow users to interactively change data visualization parameters such as column selection, segmentations, and filters require that developers express queries with a large degree of dynamic variation in query operations such as joins, aggregations, and filters.

- **How to compute the transformations?** Deciding during which phase of data processing to compute the transformations poses awkward tradeoffs. A common data warehousing approach is to rely on precomputation, maintaining a "read-friendly" denormalized copy of transactional data optimized for OLAP workloads [4]. If the denormalized copy is stale with respect to database updates, it tends to not satisfy applications with a high bar for fresh and consistent data, and the necessary offline pipelines tend to make systems more stateful and operationally complex. Transactionally updated materialized views on the other hand increase the cost of writes. Finally, computing most transformations as part of online queries makes achieving interactive (sub-second) latencies challenging, especially if queries read

*Work performed while at Google, Inc.

SIGMOD/PODS'16 June 26 - July 01, 2016, San Francisco, CA, USA

ACM ISBN 978-1-4503-3531-7/16/06.

DOI: http://dx.doi.org/10.1145/2882903.2904444

from multiple data stores with different performance characteristics.

In this paper we describe Shasta, a middleware system built at Google to power interactive reporting functionality in critical business applications. Using a combination of techniques in language and system design, Shasta addresses the challenges listed above at scale. Shasta encapsulates the transformation logic between stored data and user-facing concepts and hides it from applications by providing an abstraction similar to denormalized views over diverse data stores. To its clients, *Shasta views* are parameterized virtual read-only tables that are significantly easier to understand and query than the underlying schemas. Shasta's system components power several critical business applications at Google, including the AdWords Web UI and corresponding developer APIs. Figure 1 shows Shasta as part of the AdWords data processing architecture.

What makes Shasta quite different from common approaches to data warehousing is how its views are expressed and computed. First, Shasta views are expressed in RVL, a declarative language that combines SQL-like functionality with limited but powerful procedural constructs, designed to succinctly capture the dynamic nature of interactive applications. RVL code is dynamically compiled to SQL. Second, Shasta does not rely heavily on precomputation to minimize query time transformations, although view definitions are highly complex. Rather, many view queries "go all the way" from normalized data storage to user-facing concepts. This stateless and dynamic approach enables rich interactive UIs, with fresh and consistent data everywhere, yet without the burden of maintaining separate denormalized storage dedicated to query workloads. Shasta uses F1's distributed SQL engine to execute complex generated queries over multiple data sources. With key extensions – including a cache that leverages spatial and temporal locality of data accessed by application users – F1 runs the resulting interactive workloads reliably at low latency and scales well to more data-intensive workloads.

The contributions of this paper are:

- We introduce an integrated language and system approach to designing application middle tiers for interactive reporting applications. The approach is unusual in its heavy reliance on online data transformations.
- We show how, using a new SQL templating language, we have made the maintenance of complex business logic on top of vast legacy schemas by large engineering teams more efficient.
- We show how, thanks to F1's distributed query execution architecture, our approach achieves a high degree of operational simplicity and data freshness, keeps interactive query latencies low, and scales to more data-intensive workloads.
- We describe an implementation of this approach, our experience with it, its benefits, and its limitations.

The rest of the paper is structured as follows: Section 2 lists Shasta's system requirements, Section 3 presents the overall architecture, and Sections 4-6 describe the major components in more detail. In Section 7 we report from our experience running Shasta in production at Google. Section 8 summarizes related work.

2. SYSTEM REQUIREMENTS

The design of Shasta was driven by the following requirements of critical interactive business applications at Google:

Data stores are diverse. Generally, user-facing business applications at Google depend on multiple data stores. Data transactionally updated by users, or dimensions, is often stored in the F1 RDBMS, which is backed by Spanner storage. On the other hand,

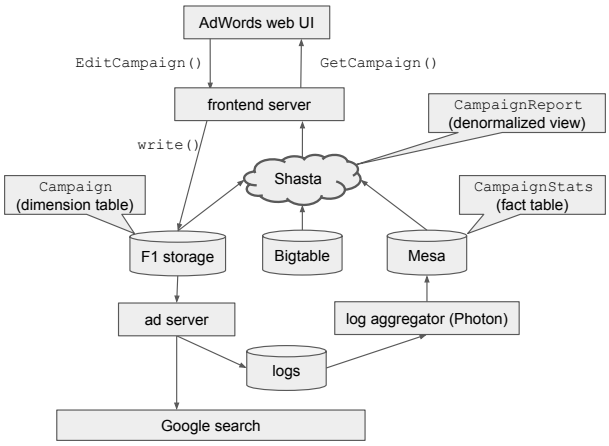

Figure 1: Simplified AdWords data-processing architecture.

performance metrics, or facts, may be stored in systems like Mesa, where they can be continuously updated by Google's internal systems. In AdWords for instance, advertiser-provided campaign configurations are stored in F1, and financially critical metrics about ad campaign performance are stored in Mesa. Completing the picture, some application state is stored in Bigtable [3] for legacy reasons. Intuitively, different data stores have different performance characteristics. Mesa supports high throughput updates by batching them and low latency reads by virtue of its sophisticated data versioning, its storage format, and other techniques [8]. On the other hand, system parameters and schemas in F1 deployments tend to be optimized more for write latency and less for read latency, in order to mitigate the overhead (50–150ms) of cross-region two-phase commits [13].

View computations are complex. For schema design and legacy reasons, there is often a large gap between the concepts shown to application users and the schemas of underlying data stores. For instance, computing the main table in the AdWords UI "Campaigns" tab involves joining and aggregating approximately 20 F1 tables and 20 Mesa tables, with intricate business logic applied to the scanned data. Furthermore, interactive reporting UIs often display multiple pivots of the same data next to each other, e.g., a table breaking down metrics per campaign, a chart showing the same metrics over time, and a single row summarizing the table metrics. In order to power such UIs efficiently and ensure that different pivots are fully consistent, shared query portions should be computed only once. Given these complexities, view computations tend to be expensive and difficult to formulate.

Views must reflect data updates immediately. To be truly interactive, applications allow users to explore performance metrics *and* modify their business data. The AdWords Web UI is a good example, allowing advertisers to explore how their campaigns are performing and update campaign configurations inline, in the same tables. Consider a scenario where a table in the UI shows that an ad campaign's budget is exhausted. In response, the advertiser may decide to increase the budget, which is easily done in the same UI table row. Any future rendering of the table must then consistently reflect the budget change. Due to the complexity of view definitions, a user changing a single cell in a sorted UI table can induce subtle changes to other columns in the same row and the application may choose to recompute the entire table immediately after the write. In AdWords, advertiser changes to campaign data result in writes to F1, bypassing Shasta. Immediately after the write, the

frontend server may request table data for the given Shasta view and expect subsequent queries to return fresh and consistent data, taking into account the most recent update by the user.

View queries from interactive UIs must be fast. Shasta queries are on the critical path to rendering pages in user-facing applications with sub-second latency targets. Importantly, interfaces and APIs in such applications are often scoped to an individual user's business data, so that typical Shasta queries process only a modest subset of data in underlying data stores. Still, individual fully optimized queries can scan gigabytes of raw data from 50 or more physical tables, and the transformations required for a query can be computationally demanding. Fortunately, interactive applications tend to be used in user sessions, allowing Shasta to leverage temporal and spatial locality of accessed data.

View definitions are parameterized. To its clients, Shasta views are parameterized virtual tables, which provide access to denormalized data while hiding complex storage schemas with hundreds of tables. Issuing a Shasta query is straightforward: The client specifies a view name, a list of column names, a simple specification for how to filter and sort resulting rows, and more ad-hoc parameters that customize the view semantics. To answer a given Shasta query, Shasta then needs to compute the view in the context of the given parameters. In practice, parameter values can have a major effect on the semantics of a view. For instance, depending on whether the user is part of a feature whitelist, a Shasta view may require data to be read from different tables and joined in a different structure.

View implementation must be manageable. In practice, hundreds of software engineers across different teams need to modify shared view definitions and schemas. Resulting changes need to be released to production multiple times per week, without downtime. In order to achieve these goals, Shasta needs a view definition format that helps developers easily reason about and modify views, and this format needs to scale to complex and highly parameterized view definitions. Ideally, view definitions should be mostly declarative, but views must be allowed to invoke some application code written in a procedural language such as C++ or Java.

3. SYSTEM ARCHITECTURE

In principle, it would be reasonable to express Shasta view transformations using SQL and leverage F1's distributed query engine for scalable query execution, since the transformations required by Shasta views are a natural fit for SQL. However, it is challenging in practice to define Shasta views in terms of SQL and meet the latency requirements of interactive applications, given the complex nature of queries. Shasta's system architecture overcomes these two challenges, thus allowing query execution to be handled by F1.

In order to push all computation to F1, Shasta needs to translate each client query into a single SQL query. This is difficult due to the complexity of data transformations, paired with a wide variety of queries corresponding to different view parameter values. The Shasta architecture simplifies the task of SQL generation: View definitions are expressed in a new language called RVL and are translated to SQL using a just-in-time compiler. The syntax of RVL is SQL-like, but extended with higher-level constructs which make it easier to define Shasta views concisely while accounting for dynamic view parameters. RVL supports user-defined functions (UDFs), allowing Shasta view definitions to invoke procedural application code.

The following extensions to the F1 query engine were made to meet the needs of Shasta:

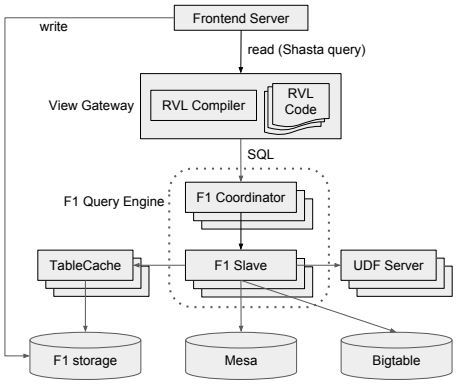

Figure 2: Shasta architecture.

- We improved support for UDFs in F1. Specifically, we added a component called UDF server to F1, which supports safe evaluation of compiled C++ and Java UDFs with high throughput.

- In order to meet the latency constraints of Shasta, we added an in-memory read-only cache between F1 storage and query engine, called TableCache. Shasta uses TableCache to access consistent snapshots of F1 data significantly faster than reading from F1 storage directly. The cache is kept up-to-date using a light-weight protocol based on F1's Change History feature. For Shasta applications, maintaining similar caches for other data sources such as Mesa and Bigtable has proven either less critical or too expensive in practice.

Figure 2 shows how the components of Shasta fit together to answer queries. A query issued by the application's frontend server specifies a view name, columns to query, and view parameters such as application user ID or feature ID. In addition, the client often specifies a timestamp, which tells Shasta to use F1 data which is consistent with the snapshot at that timestamp. Exposing the list of available view schemas to application developers is typically done via shared code repositories. The *view gateway* receives the Shasta query and invokes the RVL compiler with view parameter bindings to generate a potentially large SQL string. The F1 query engine executes the generated SQL, using UDF servers to evaluate any UDFs in the query. TableCache accelerates access to F1 storage during query execution, while the query engine reads directly from data sources other than F1 such as Mesa and Bigtable. F1 supports access to such "external" data sources using a plugin API [13], allowing for tight integration with the query engine. F1 pushes filters and projections to external data sources where possible, performing other computations itself. Optimizations that would delegate more complex query operations to external data sources [9] are left as future work.

It is worth noting that Shasta does not rely on precomputation and materialization of intermediate view query results. Although precomputation may improve the latency of Shasta queries, it tends to be a poor fit for Shasta applications, especially when dealing with financial user data: Keeping materialized results synchronized with user-issued changes to underlying data stores makes writes more expensive, which is often not an option. It would also be impractical to use stale materializations while satisfying freshness requirements, due to the complexity of Shasta views. These drawbacks can be avoided by evaluating queries on raw data.

The following sections provide more details on RVL (Section 4), Shasta's use of the F1 query engine and UDF servers (Section 5), and TableCache (Section 6).

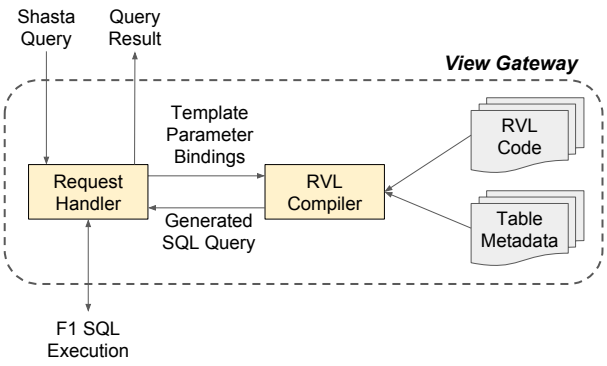

Figure 3: Using the RVL compiler in Shasta.

4. RVL: RELATIONAL VIEW LANGUAGE

As described in Section 3, Shasta uses a language called RVL (Relational View Language) to define Shasta views, and compiles RVL into equivalent SQL which can be executed in the F1 query engine. RVL helps view definition authors focus on the specifics of each view and to not concern themselves with the problem of generating SQL. The features of RVL can be summarized as follows:

- At the core of RVL is a self-contained query language. The syntax of the query language is similar to SQL, but with important differences in semantics. In particular, RVL uses information in the schema to automatically aggregate data when columns are projected, which makes it easier for developers to express the complex aggregation logic of Shasta views.
- RVL embeds its query language in higher-level constructs called view templates. A view template specifies a dynamic query using replaceable parameters. Within each template, a query can be built as a sequence of subquery assignment statements. Each subquery in a template may refer to template parameters and instantiate other view templates. Overall, composable templates and sequential query construction allow a view's implementation to be factored into manageable and reusable pieces.

RVL code is structured as a collection of view templates. In order to actually generate SQL, the RVL compiler requires one template to be specified as an entry point, along with parameter bindings for that template. Figure 3 illustrates the role of the RVL compiler in the view gateway of Shasta. When Shasta receives a query, the request handler first translates that query into parameter bindings to pass to the RVL compiler, which returns generated SQL. The request handler then executes the SQL query on F1 and forwards the query results back to the Shasta client.

In this section, we provide an overview of RVL. We start by describing the query language in isolation, and then show how the query language is embedded in view templates. We next describe the most interesting aspects of RVL compilation and conclude with an end-to-end RVL example.

4.1 Query Language

RVL includes a query language to specify data transformations. The syntax and semantics are similar to SQL, with one fundamental difference: RVL automatically determines how to aggregate query results. As we show in examples below, automatic aggregation can help simplify the specification of view queries for reporting applications built on Shasta. To take advantage of automatic aggregation, the metadata of each column may optionally specify an *implicit aggregation function*. If a column has an implicit aggregation function, we refer to that column as an *aggregatable column*. Otherwise, it is a *grouping column*. When a query operation is performed in RVL, the result will only preserve the unique values of the grouping columns, and the aggregatable columns are implicitly aggregated for each row based on the associated aggregation function. In other words, the grouping columns always form a unique key for a relation in RVL, and the aggregatable columns represent measures associated with values of the unique key. In the special case where all columns are grouping columns, the behavior of RVL is consistent with relational algebra on sets. In practice, Shasta applications specify implicit aggregation functions as overlays on an existing database schema and include this information in the table metadata given to the RVL compiler. RVL also supports syntax to modify or remove implicit aggregation functions assigned to the columns of a relation (see Section 4.4).

RVL queries may use several syntactic constructs borrowed from SQL such as SELECT, FROM, WHERE, JOIN, and UNION. The behavior of these operations is similar to SQL, except that implicit aggregation is applied after each operation to yield a set of unique rows. There is no GROUP BY clause in RVL, since aggregation is implicit for each SELECT. RVL also makes extensions to SQL in order to help with large-scale code maintenance, as described in Section 4.2.

We can illustrate the behavior of RVL queries on the following two tables representing information about employees and buildings of a company:

Employee table:

EmpId	*DeptId*	*BldgId*	*Salary* [SUM]
I	A	X	20
J	A	Y	30
K	B	Y	40
L	B	Z	50

Building table:

BldgId	*CityId*	*Capacity* [SUM]
X	M	100
Y	N	200
Z	N	300

The columns for salary and capacity have the [SUM] annotation to indicate that they use SUM as their implicit aggregation function.

An application may need to derive the total salary or capacity for a variety of grouping columns. Using RVL, we can start by combining the tables with a left join, and let `Q0` denote this subquery:

```
Q0 = SELECT *
     FROM Employee LEFT JOIN Building USING (BldgId);
```

This provides a list of employees and information about their building, if known. We may use this subquery to write a very simple query for the total salary in each department:

```
Q1 = SELECT DeptId, Salary FROM Q0;
```

Evaluating `Q1` will return the unique DeptId values and automatically compute the sum of salaries in each department, yielding the final result:

DeptId	*Salary* [SUM]
A	50
B	90

This could be expressed in SQL using an explicit SUM and GROUP BY clause:

```
SELECT DeptId, SUM(Salary) FROM Employee
GROUP BY DeptId;
```

The RVL compiler will actually generate this SQL for `Q1`. The join with the Building table can be pruned since none of the Building

columns are required. Join pruning is a key feature of RVL, which we discuss in Section 4.3.

Implicit aggregation becomes more interesting when data is requested from both tables. For example:

```
Q2 = SELECT CityId, Salary, Capacity FROM Q0;
```

This will return the unique CityId values and automatically compute the total salaries of employees in each city, along with the capacity of the buildings in that city:

CityId	*Salary* [SUM]	*Capacity* [SUM]
M	20	100
N	120	500

The RVL compiler generates the following SQL representation of `Q2` which adds two GROUP BY steps:

```
SELECT CityId, SUM(Salary) AS Salary,
       SUM(Capacity) AS Capacity
FROM
  (SELECT BldgId, SUM(Salary) AS Salary
   FROM Employee GROUP BY BldgId)
  LEFT JOIN Building USING (BldgId))
GROUP BY CityId;
```

Observe that the inner subquery removes EmpId and DeptId, in order to aggregate the remaining columns of the Employee table before the join. The inner aggregation ensures that the capacity is computed correctly. If we naively removed all unwanted columns after the join and performed the aggregation in a single step, the capacity values would be multiplied by the number of employees in each building. Our desired result should only count the capacity of each building once. We will revisit this example and the RVL compiler's strategy for arranging GROUP BY clauses in Section 4.3.

The value of RVL can be seen by comparing `Q1` and `Q2` to the corresponding SQL representations. In RVL, we can define a single subquery `Q0` such that `Q1` and `Q2` can be expressed as simple projections over `Q0`. In contrast, the SQL representations of `Q1` and `Q2` have drastically different structure. RVL also makes it easy to derive meaningful aggregate values from `Q0` for many other combinations of grouping columns. A direct SQL representation of all possible projections over the join would need to account for all the potential arrangements of GROUP BY clauses and invocations of aggregation functions. In practice, real Shasta queries can be more complex, requiring dozens of tables to be joined. It becomes challenging for developers to formulate the correct aggregation semantics using SQL directly whereas formulating queries using RVL makes it much more simple and intuitive.

4.2 View Templates

As described above, the RVL query language provides implicit aggregation to help developers express aggregation semantics when the set of requested columns is not fixed. However, implicit aggregation does not solve all the challenges of implementing Shasta views. In particular:

- The parameters of a Shasta view may have a large impact on the RVL query. For instance, the view parameters may change the tables used in joins or the placement of filters in the query. RVL needs to be more dynamic in order to capture this wide range of possible query structures.
- An RVL query could grow quite large with many column transformations and deeply nested joins. A typical Shasta view would require 100s of lines of code, and expressing that as a single large query can be difficult to read and maintain.

RVL *view templates* solve these problems. View templates allow large queries to be constructed dynamically from smaller pieces which can be composed and reused by multiple Shasta views. A view template takes input parameters which are used to represent the dynamic aspects of a query (e.g., list of requested columns), and returns a corresponding RVL query using the parameter values. In other words, for any fixed choice of parameter values, a view template is shorthand for an RVL query (similar to the traditional notion of a view).

A view template may be referenced in the FROM clause of RVL queries by passing values for its input parameters, and the reference will be replaced with the query generated by the view template. The following values may be bound to view template parameters:

- *RVL text:* A string containing valid RVL syntax can be bound to a view template parameter, and that parameter can be referenced in places where it would be valid to inject the RVL syntax string. For example, a template parameter bound to the string `"X,Y"` could be referenced in a SELECT clause, and the template parameter reference will behave exactly as if `"X,Y"` were written directly in the query.
- *Nested dictionary:* A template parameter can be bound to a dictionary of name-value pairs, where the values can either be another dictionary, or RVL text. Intuitively, a nested dictionary is a collection of RVL text parameters with hierarchical names.
- *Subquery:* A template parameter can be bound to an RVL subquery, and referenced anywhere a table can be referenced. A subquery value differs from RVL text values, in the sense that subquery values are substituted in a purely logical manner which is independent of the syntax used to create the subquery. In contrast, an RVL text value is purely a text injection, which allows any variables in the RVL text to be interpreted based on the context where the parameter is referenced.

RVL text values allow RVL parameters to be more flexible than traditional SQL runtime parameters, since RVL allows parameters to represent large substructures of queries. However, RVL text values do not allow for arbitrary code injection. In order to make view templates less error-prone, an RVL text value is only allowed to contain a few specific syntactic forms, such as scalar expressions, expression lists, and base table names. There are also strict rules controlling the locations where each syntactic form can be substituted.

We use the following example to illustrate the basic template syntax and semantics:

```
view FilterUnion<input_table, params> {
  T1 = SELECT $params.column_name FROM $input_table;
  T2 = SELECT $params.column_name FROM Employee;
  T = T1 UNION T2;
  return SELECT * FROM T
         WHERE $params.column_name >= $params.min_value;
}
```

The view template contains three assignment statements which give aliases to subqueries, and the fourth statement returns the final query. There are two template parameters:

- `input_table` can be bound to a table name or subquery. It is referenced in the first FROM clause as `$input_table`.
- `params` must be bound to a nested dictionary. In this example, `$params.column_name` should be the name of a column in `$input_table`, and `$params.min_value` is a lower bound that we want to apply to that column.

The behavior of the view template is intuitive: The first two statements fetch a dynamically chosen column from the `$input_table` parameter as well as the Employee table, the third statement combines the two sets of values, and the final statement applies a lower bound to the values before returning them.

A view template can be designated as an entry point in the RVL code, in which case it is called a *main view template*. RVL provides an API to invoke a main view template, with a nested dictionary as a parameter. The main view template can use one or more output statements to specify the desired result. For example, using the previous FilterUnion view template:

```
main OutputValues<params> {
  b = SELECT * FROM Building;
  all_values = SELECT * from FilterUnion<@b, $params>;
  output all_values AS result;
}
```

The output statement specifies a table to produce as the final result when the main view template is invoked, as well as an alias for that table. If there are multiple output statements, the aliases must be unique so that the Shasta view gateway can distinguish the results. Multiple output statements can reference the same RVL subquery by name which is useful when applications need to display multiple pivots of the same shared view computation. To achieve consistency between different data pivots within a view query, RVL guarantees that each named subquery is only executed once.

In Section 4.4 we present a larger RVL example which uses additional syntax features. Most interestingly, RVL view templates may use control structures written as if/else blocks to dynamically choose between two or more subqueries.

4.3 RVL Compiler and Query Optimization

The RVL compiler generates SQL which will produce the output tables specified by a main view template, given the required parameter bindings. Shasta executes the generated SQL using the F1 query engine, taking advantage of F1's query optimizations and distributed execution. In order to perform SQL generation, the RVL compiler first resolves references to view templates and named subqueries, producing an algebraic representation of an RVL query plan that includes all outputs of the invoked main view template. The RVL compiler performs some transformations to optimize and simplify the query plan before translating it to SQL. In this section, we describe some details of RVL query optimization and explain why it is important.

The RVL compiler optimizes query plans using a rule-based engine. Each rule uses a different strategy to simplify the plan based on algebraic structure, without estimating cost. In practice, rule-based optimization is sufficient because the only goal is to simplify the generated SQL, rather than determine all details of query execution. We avoid using cost-based optimization because a cost model would tie the RVL compiler to a specific SQL engine and make it less generic.

The intuitive reason to optimize an RVL query plan before generating SQL (as opposed to relying on F1 for all optimizations) is to take advantage of RVL's implicit aggregation semantics. Several optimization rules are implemented in the RVL compiler relying on properties of implicit aggregation to ensure correctness. The RVL compiler also implements optimization rules which do not depend directly on implicit aggregation, because they interact with other rules that do depend on implicit aggregation and make them more effective. We describe a few of the more interesting rules below.

Column Pruning: Recall the Employee/Building example earlier in this section. Our SQL representation of `Q2` performed a projection and aggregation before the join, which differs from the order of operations in the RVL for `Q2`. The join and aggregation steps are reordered by an RVL optimization rule called *column pruning*. Without column pruning, the equivalent SQL representation of `Q2` would be:

```
SELECT CityId, SUM(Salary) AS Salary,
       SUM(Capacity) AS Capacity
FROM
  (SELECT BldgId, CityId,
          SUM(Salary) AS Salary, Capacity
   FROM Employee LEFT JOIN Building USING (BldgId)
   GROUP BY BldgId, CityId, Capacity)
GROUP BY CityId;
```

Observe that this SQL representation performs two stages of aggregation after the join, in order to compute correct aggregate values for both salary and capacity. A sufficiently advanced SQL optimizer may be able to optimize this query by pushing the inner aggregation below the join, but this is a difficult optimization to generalize in the context of SQL [16]. For larger RVL queries, the SQL representation may become much more complicated when computing aggregate values after joining. In the worst case, an implicit aggregation may require aggregatable columns to be computed in a temporary table and joined back into the original query. That pattern in particular is extremely difficult for a SQL engine to optimize, so column pruning is needed in order to simplify the task of the F1 query optimizer. Moreover, the logic for pruning columns in the RVL compiler is straightforward due to implicit aggregation semantics. For all these reasons, the RVL compiler aggressively prunes unneeded columns and performs aggregations before joins whenever possible.

Filter Pushdown: In a SQL database, filter pushdown could be considered one of the most basic optimizations, where the goal is to filter data as early as possible in the query plan. At first glance, it might seem unnecessary for RVL to push down filters, since the F1 query optimizer is fully capable of performing this optimization. However, filter pushdown can improve the effectiveness of the column pruning optimization. For example, if there is a filter on a column which is not part of the final result, the filter will prevent column pruning from removing the column before the filter is applied. It is crucial for the filter to be pushed down as early as possible in the query plan, so that the column can be pruned early as well.

Left Join Pruning: In RVL, if a left join does not require any of the columns from its right input, the right input can be removed from the query plan. In a general SQL database, this optimization is less obvious and less likely to apply, since a left join may duplicate rows in the left input. Consider the following example of a SQL left join, using the example Employee and Building tables:

```
SELECT EmpId, SUM(Salary)
FROM Employee LEFT JOIN Building USING (BldgId);
```

If it is known that each Employee row will join with at most one row in the Building table, the join can be pruned, resulting in:

```
SELECT EmpId, SUM(Salary) FROM Employee;
```

A SQL optimizer might be able to perform this optimization if it knows that BldgId is a unique key of the Building table. The optimization would become more difficult to do if the Building table were replaced with a complex subquery. On the other hand, the left join in RVL is trivial to prune, since the inputs of the join are always guaranteed to be sets, and the column pruning optimization will prune all Building columns except BldgId.

The join pruning optimization makes RVL programming more convenient. A user can add many left joins to their view templates to fetch columns which might not be required, and they can be confident that the RVL compiler will know which joins can be skipped.

4.4 Example View Template

Figure 4 shows example schemas for F1 and Mesa tables, using advertising concepts from the AdWords application. The example

Customer(root table)

CustomerId	CustomerInfo
20	{ name: "flowers" }

BudgetSuggestionV1

CustomerId	**BudgetId**	SuggestionInfo
20	200	{ suggested_amount: 120 }

BudgetSuggestionV2

CustomerId	**BudgetId**	SuggestionInfo
20	200	{ suggested_amount: 118 }

Campaign

CustomerId	**CampaignId**	CampaignInfo
20	100	{ name: "Rose" status: "ENABLED" budget_id: 200 }
20	101	{ name: "Tulip" status: "ENABLED" budget_id: 200 }
20	102	{ name: "Daisy" status: "ENABLED" budget_id: 201 }

Budget

CustomerId	**BudgetId**	BudgetInfo
20	200	{ amount: 100 }
20	201	{ amount: 50 }

CampaignStats

CustomerId	**CampaignId**	**Device**	Impressions	Clicks	Cost
20	100	'Desktop'	20	5	3
20	100	'Tablet'	10	3	1
20	101	'Mobile'	30	4	2
20	102	'Desktop'	40	10	5

CampaignConversionStats

CustomerId	**CampaignId**	**Device**	**ConversionType**	Conversions
20	100	'Desktop'	'Purchase'	2
20	100	'Desktop'	'Wishlist'	1
20	101	'Mobile'	'Wishlist'	2

Figure 4: Example storage schema. Bolded column names refer to primary keys.

```
view CampaignDimensions<params> {
  campaign = SELECT *,
    CampaignInfo.name AS Name,
    CampaignInfo.budget_id AS BudgetId
    FROM Campaign;
  budgets = SELECT *,
    BudgetInfo.amount AS BudgetAmount
    FROM Budget;
  budget_suggestion_table =
    if ($params.use_budget_suggestion_v2) {
      BudgetSuggestionV2;
    } else {
      BudgetSuggestionV1;
    }
  budgets_with_suggestion = SELECT *
    FROM budgets LEFT JOIN budget_suggestion_table
    USING CustomerId, BudgetId;
  return SELECT *,
    ComputeCampaignStatus(CampaignInfo, BudgetInfo,
                          BudgetSuggestionInfo) AS Status
    FROM campaign LEFT JOIN budgets_with_suggestion
    USING CustomerId, BudgetId;
}
view CampaignFacts<params> {
  return SELECT *,
    MakePair(Impressions, Clicks)
      AS ClickThroughRate [aggregation = "RateAgg"]
    FROM CampaignStats FULL JOIN CampaignConversionStats;
}
main CampaignReport<params> {
  campaign_report = SELECT $params.main_table_columns
    FROM CampaignDimensions<$params>
         LEFT JOIN CampaignFacts<$params>
         USING CustomerId, CampaignId;
  output SELECT *
    FROM campaign_report
    WHERE $params.filters
    ORDER BY $params.order_by_columns
    LIMIT $params.limit as top_k_table;
  output SELECT $params.summary_columns
    FROM campaign_report as summary;
}
```

Figure 5: CampaignReport RVL code.

uses F1 dimension tables Customer, Campaign, Budget, BudgetSuggestionV1, and BudgetSuggestionV2, using CustomerId as the root ID BudgetSuggestionV1 and BudgetSuggestionV2 tables capture a case that often comes up in practice: The application is migrating from an older to a newer, higher quality representation of budget suggestions. The application may want to use the new suggestions for only a small whitelist of advertisers initially, and ramp up slowly to all customers. We therefore maintain both versions during the transition period. Since different teams maintain the respective backend pipelines, separate tables help clarify ownership.

The example also uses Mesa fact tables CampaignStats and CampaignConversionStats. Impressions, Clicks, Cost, and Conversions columns use SUM for implicit aggregation. CampaignConversionStats is a separate table because conversions can be broken down by the additional dimension ConversionType.

Abstracting over the complex storage schema, Shasta exposes a flat denormalized view CampaignReport. The view schema exposes the following columns: CustomerId, CampaignId, Name, Status, BudgetAmount, Device, Impressions, ClickThroughRate, Clicks, and Conversions. RVL code for CampaignReport is shown in Figure 5. The following aspects are worth noting:

- The CampaignDimensions view template performs a join of F1 tables. The input parameter `use_budget_suggestion_v2` indicates which version of budget suggestion to use. The Status column is computed with a user-defined function (UDF).
- The CampaignFacts view template performs a join of Mesa tables. ClickThroughRate is made explicitly aggregatable by a user-defined aggregation function (UDAF) RateAgg. No explicit GROUP BY is specified, as the RVL compiler generates the correct GROUP BY clauses based on the request context.
- The main view template CampaignReport joins data from CampaignDimensions and CampaignFacts. The two view outputs `top_k_table` and `summary` share the `campaign_report` subquery, ensuring consistency of the two data pivots.
- `params.filters` typically contains a filter on CustomerId, but it can also contain filters on aggregatable columns like Impressions. The RVL code filters the data after projecting the final columns, but the RVL compiler may move filters to be applied earlier when possible.

Figure 6 illustrates the entire flow of a sample Shasta query with (i) a specific Shasta query issued by a Shasta client against the "CampaignReport" Shasta view, (ii) RVL template parameters generated from the Shasta query by the view gateway, and (iii) the Shasta query result. Note the result contains only two campaigns because of the "limit: 2" clause in the query, while the summary row contains aggregate stats for all campaigns.

5. QUERY EXECUTION ON F1

F1's SQL query engine has been described in [13]. It supports both *centralized* and *distributed* execution of queries. Centralized execution can deliver low latencies for simple queries with input data volumes that are easily processed on a single F1 server, and distributed F1 queries are a natural fit for workloads with larger in-

Shasta query

```
view_name: CampaignReport
columns: CampaignId, Name, Status, Budget, Impressions,
         Clicks, ClickThroughRate, Conversions
sort order: Clicks Descending
limit: 2
summary: on
application_params: CustomerId=20, BudgetSuggestionVersion=2
```

RVL template parameters

```
main_table_columns: CampaignId, Name, Status, BudgetAmount,
                    Impressions, Clicks, ClickThroughRate, Conversions
filters: CustomerId = 20
order_by_columns: Clicks DESC
limit: 2
summary_columns: Impressions, Clicks, ClickThroughRate, Conversions
use_budget_suggestion_v2: true
```

Query result

CampaignId	Name	Status	BudgetAmount	Impressions	Clicks	ClickThroughRate	Conversions
102	Daisy	Enabled	50	40	10	0.25	0
100	Rose	BudgetThrottled	100	30	8	0.27	3
Summary				100	22	0.22	5

Figure 6: Shasta view invocation.

puts and less stringent latency requirements, e.g., queries scanning all rows in a large F1 or Mesa table. Shasta represents a novel use of F1's query engine as it places heavily distributed query execution at the heart of latency sensitive and business-critical applications. While Shasta queries are typically scoped to a modest subset of data in underlying databases (e.g., a single advertiser's campaign configurations and performance metrics), the combination of remote input data, diverse data stores, data freshness requirements, and complexity of query logic make it challenging to achieve low latencies reliably using centralized execution, in Google's production environment based on shared commodity hardware. In the remainder of this section, we describe core F1 query engine features that were critical for the Shasta workload, our approach to supporting user-defined functions in F1, and how we tuned the distributed query processing pipeline for low latency.

5.1 Engine Features Critical for Shasta

In distributed F1 query execution, a *query coordinator* receives the incoming SQL query. The query is parsed and an execution plan is laid out as a directed acyclic graph (DAG) of potentially hundreds of *plan parts*. Each plan part is a tree of operators like scan, filter, project etc. Data flows from the leaves of the DAG towards the coordinator. F1 aims to maximize streaming of data between operators and to minimize pipeline stalls. Each plan part is annotated with the degree of parallelism at which it is to be run. The coordinator schedules plan parts to run on *F1 slaves* and configures the data flow between them as per the plan's DAG structure.

Hash-based repartitioning: The F1 query engine was designed for settings where co-partitioning of data between storage and query processing is not feasible, and where storage systems such as Spanner apply effectively arbitrary partitioning of data. To this end, F1 supports hash-based repartitioning of data during query execution. This support is critical for Shasta workloads, where input data stores are diverse and often not read-optimized. Hash-based repartitionings are used for co-partitioining in joins and for distributed aggregations. Shasta prefers to arrange large numbers of joins and aggregations in shallow, bushy trees to reduce latency and to allow for increased parallelism. As a result, latency depends more on tree depth than on the number of joins. Using SQL hints, skew-sensitive joins to small tables are often executed by broadcasting the smaller input, leaving the distribution of the larger input unchanged.

DAG-structured query plans: It is idiomatic in RVL to assign variable names to subqueries and then refer to them multiple times in subsequent RVL code. For instance, an RVL template may output a table and a summary row as different pivots of the same named subquery. For data consistency and computational efficiency, the underlying query engine should compute the subquery only once and feed results into multiple downstream computations. F1 supports such common table expressions using standard SQL WITH syntax. F1 internally represents multiple references to the same subquery as a DAG structure in which the subquery has multiple parent operators. Buffering mitigates the case of parent operators consuming result rows from the shared subquery at different rates.

External Data Sources: Shasta queries commonly scan and join data from storage systems as diverse as Spanner, F1 TableCache, Mesa, and Bigtable. Using a plugin framework for federated querying, the F1 query engine supports these data sources and several others. Central to the plugin framework is the abstraction of a *ScanOperator*. For every supported data source, a ScanOperator implementation translates between the data source API and the F1 SQL runtime. ScanOperator implementations can be carefully tailored to peculiarities of each data store. The resulting deep integrations between query engine and data stores were key in meeting the latency and scalability requirements in Shasta query workloads. A prime example for this is how F1 determines the degree of parallelism for distributed table scans, a fundamental feature of distributed query processing. The appropriate degree of parallelism for a table scan depends on how much data the table scan is expected to return, which in turn depends on the given query and data source. For each supported data source, the F1 ScanOperator plugin implements a GetPartitions() function. The F1 query planner calls this function to retrieve cardinality information in the context of a specific query. GetPartitions() implementations are naturally tailored to each data store. For instance, the Mesa ScanOperator's GetPartitions() implementation leverages precise Mesa-internal metadata only accessible by issuing Mesa RPCs, while the GetPartitions() implementation for TableCache uses metadata stored by an offline job in F1 storage. Compared to F1 maintaining cardinality metadata for all data sources centrally, the GetPartitions() protocol allows for more precise and up-to-date cardinality information. Federated ownership of metadata for different data stores also tends to be more scalable operationally.

5.2 UDF Servers

RVL code can invoke user-defined functions (UDFs) written in procedural languages such as C++ or Java. Motivations include sharing critical procedural code between RVL definitions and other systems processing the same data, re-use of subtle legacy code, and expressivity. UDFs are a common feature in SQL engines. However, UDF support in F1 allows users to deploy compiled C++ or Java UDF code in custom binaries called UDF servers that run in production alongside the F1 query engine, as separate processes and under client teams' credentials. In order to be able to register with the F1 query engine, UDF server binaries must implement a

standard RPC API defined by F1. F1 provides "glue" libraries that make building a compliant UDF server trivial, given just a library of C++ or Java functions written by the client team. During execution of queries with UDF calls, F1 slaves issue RPCs to appropriate UDF servers, which in turn invoke corresponding UDF code. Compared to dynamically loading UDF libraries in F1 processes, this model provides the same release schedule flexibility but significantly stronger system isolation and security properties: UDF code never runs using F1's credentials and the F1 SQL runtime is protected from crashes or memory corruptions introduced by bugs in C++ UDFs, and from garbage collection spikes or exceptions introduced by Java UDFs.

One challenge in this decoupled approach is to minimize the latency costs that often come with turning function calls into RPCs. We have managed to largely mitigate these costs using two techniques. First, F1 slaves and UDF servers have independent degrees of parallelism: While F1 slaves use a single thread to process each plan part, an F1 slave thread calling a UDF can distribute rows across hundreds of UDF servers. Second, matching other parts of the F1 query engine, we heavily leverage batching and pipelining for UDF calls: As rows stream through the CallUdf operator, the operator maintains a queue of UDF server RPCs and blocks on results only if more than a certain threshold of RPCs are in-flight. RPC results are processed out-of-order if possible, reducing the impact of slow RPCs in a batch and hence reducing tail latency.

5.3 Distributed Execution Improvements

We highlight two distributed execution improvements that were key in meeting Shasta's low latency and scalability requirements.

Query Planning: During query planning, the query coordinator needs to determine the degree of parallelism for each plan part. Choosing this parameter well is key in scaling queries to different input sizes. Using GetPartitions() as explained above, we determine the degree of parallelism for scan plan parts (i.e., leaves), aiming to assign each scan partition to a different slave. For non-leaves, parallelism information is propagated up the DAG. For each plan part, the degree of parallelism is set to the maximum value among its children. This ensures that each internal plan part's parallelism takes input data size into account, improving scalability.

Query scheduling: After query planning, the coordinator assigns plan parts to specific F1 slave jobs and connects them with each other as sources and sinks per the query plan's DAG structure. Common Shasta queries have hundreds of plan parts. Instead of delaying the start of query execution until all plan parts are scheduled, scheduling is done in a bottom-up fashion: Leaf plan parts are scheduled first and can start working while their sink plan parts are still being scheduled. Once the sinks are scheduled, leaf parts are updated to use the respective sinks. This approach is used all the way up the DAG. By interleaving scheduling and execution in this way, we reduced query latency by tens of milliseconds.

6. F1 TABLECACHE

TableCache is a distributed in-memory read-through cache for data stored in F1, situated between F1 storage and the F1 query engine. As Shasta applications access data in per-user sessions, TableCache is designed for workloads where individual queries are scoped to one specific root ID. In F1, a *root ID* is the primary key of the root in a hierarchy of tables, and application users are identified by a root ID in typical F1 schemas. Given (table, root ID, timestamp) triples, TableCache serves data to the F1 query engine faster and at significantly higher throughput compared to raw F1 storage. For advertiser sessions in the AdWords Web UI for example, TableCache often achieves 20 times higher read throughput. TableCache is multi-versioned and provides the same snapshot read semantics as F1, but only for a moving window of timestamps from "just committed" to a few minutes in the past. TableCache does not support reads across root IDs or for timestamps further in the past.

The intuition behind TableCache is that, for F1 storage backed by Spanner, there is a natural tension between optimizing the database for reads vs. for writes: write latencies tend to benefit from keeping the number of shards per Spanner directory modest, to minimize participants in transactions. Read throughput on the other hand benefits from a larger number of smaller shards. A distributed read-only in-memory cache can avoid this tension: Read throughput can be dramatically increased by serving data from RAM and by using shard sizes and data structures solely optimized for reads. However, as important Shasta applications cannot tolerate stale data with respect to recent user updates, the cache must be kept consistent and fresh without significant negative impact on write or read latencies. TableCache achieves this using F1 Change History.

The following subsections describe TableCache's design in three parts. First, we divide F1 rows for (table, root ID) pairs into small table shards using periodic offline analysis. Second, a set of potentially hundreds of distributed in-memory cache servers lazily load and evict table shards. The cache server API is designed to work well as a data source for distributed F1 SQL queries, and a hash-based protocol balances table shards across cache servers. Finally, loaded table shards are versioned based on F1 timestamps. If a shard is not fresh enough for a given query, we update it by incrementally applying changes from F1 Change History.

6.1 Sharding of Cached Data

To allow for more read parallelism compared to reading from F1 directly, we divide F1 rows for a given (table, root ID) into *table shards* significantly smaller than shards at the F1 storage level, often 10x smaller. A periodic offline job stores TableCache sharding metadata in an F1 system table, mapping each (table, root ID) pair to a list of table shard boundary keys and size estimates. The sharding algorithm aims to create small yet evenly sized shards, maximizing read throughput yet minimizing input data skew during F1 query processing. The offline job reruns its analysis for root IDs with a significant number of recent changes, which can be determined easily in F1 using a Change History SQL query. We describe TableCache's use of F1 Change History in more detail later in this section.

6.2 Cache Serving

TableCache was designed to be used as a data source in distributed F1 SQL queries such as the following:

```
DEFINE TABLE tablecache_Campaign(format='tablecache',
                                 table_name='Campaign');
SELECT * FROM tablecache_Campaign WHERE CustomerId = 20;
```

The F1 query engine works best with external data sources that support separate RPCs for dynamic cardinality estimation and for reading data. TableCache servers therefore expose two RPCs. Both are called from within the F1 ScanOperator implementation – i.e., query engine plugin – for TableCache.

- GetPartitions(table, root_id) returns to the F1 coordinator a list of shard split points.
- Read(table, root_id, read_timestamp, shard_id) returns to an F1 slave an ordered list of F1 rows for one table shard.[1]

Table shards are not statically mapped to cache servers – any cache server will readily load any table shard for which it receives a re-

[1]In practice, F1 can push down additional filters to TableCache.

quest. Rather, the client (the F1 query engine) is responsible for distributing table shards evenly across cache servers, by issuing Read() requests for specific shards to specific cache servers. To decide which server to talk to, TableCache clients use the deterministic hash based function GetServer which maps a table shard (table, root_id, shard_id) to a specific cache server.[2] GetServer tends to return different cache servers for different table shards, balancing table shards evenly across cache servers. Using the (table, root_id) pair (Campaign, 20) from the SQL query above and assuming the query client requests data as of F1 snapshot timestamp T_{Query}, a TableCache scan is executed in F1 as follows.

The F1 coordinator issues a GetPartitions RPC to the cache server determined by GetServer(Campaign, 20, 0). Note shard_id is always set to 0 for GetPartitions calls. The cache server looks up and returns shard split points stored by the offline job in F1 for (Campaign, 20). The coordinator schedules one F1 slave to issue a Read RPC for each table shard as determined by GetPartitions. Each slave then issues the RPC Read(Campaign, 20, T_{Query}, shard_id) to the cache server determined by GetServer(Campaign, 20, shard_id). The cache servers now need to load the requested table shards, if not yet present in RAM. Before loading, each server confirms that there is enough RAM available or evicts other table shards, using an LRU-based replacement policy. The server then reads the CampaignId range corresponding to the shard split points for shard_id from F1.Campaign at timestamp (T_{Query} − 15 minutes). "15 minutes" is a system parameter, configuring the moving window of consistent data versions that TableCache supports. The server stores read F1 rows in a modified B-tree data structure and records the checkpoint $T_{\text{Cache}} = T_{\text{Query}} - 15\text{m}$ to mark how fresh the cache for this table shard currently is. Clearly the cache is not fresh enough yet to answer queries at timestamp T_{Query}. The process of updating the cache immediately after loading a table shard or later in a session is essentially the same, and we describe that next.

6.3 Updating the Cache

TableCache heavily leverages F1's Change History feature as described in [13]. We give a brief overview of the feature before explaining how data loaded in TableCache is kept up-to-date.

Unlike traditional DBMSs, F1 provides a user-queryable change log called Change History. This log is maintained by the database system itself, not by application business logic, so it is guaranteed to capture all changes to change-tracked tables, including manually applied emergency data changes. Whether an F1 table is change-tracked or not is configured in the schema. Every F1 transaction that writes to a change-tracked table creates one or more ChangeBatch Protocol Buffers, which include the primary key and before and after values of changed columns for each updated row. These ChangeBatch records are then written to a Change History table, the primary key of which includes the associated root ID and transaction commit timestamp. Change History tables are first-class tables in F1, meaning they are tunable in the schema and queryable by F1 clients like any other F1 table. Based on the primary keys in Change History tables, clients can process Change History entries strictly ordered by root ID and commit timestamp of the corresponding F1 transactions and build a precise image of F1 data physically outside F1. This can be done using a relatively simple protocol based on checkpoints and before/after values contained in Change History records. A good example for this is how F1 TableCache keeps loaded data up-to-date.

[2]In practice, GetServer returns a specific permutation of the list of available cache servers, and clients iterate through the list, for failover in case of unhealthy cache servers.

Whenever a loaded table shard's timestamp T_{Cache} is older than the requested timestamp T_{Query}, the cache server updates the table shard by querying F1 Change History at snapshot time T_{Query}, retrieving all change records for (table, root ID) with a commit timestamp > T_{Cache}. Our system and schema are optimized to make this read fast enough to be placed on the latency critical path of interactive applications. For every returned change record, if the changed CampaignId is in range for the given table shard, the cache server applies the change to its in-memory representation, but preserving older versions within the configured moving window of snapshots supported. Once all changes are applied, the table shard's checkpoint is updated to T_{Query}, and the read request can be satisfied.

Using this protocol, TableCache achieves a powerful property: As long as queries are grouped into user sessions by root ID so that the cost of loading table shards from F1 storage is amortized over subsequent cache reads, TableCache can provide an order of magnitude higher read throughput to an application that expects 100% fresh data in requests issued immediately after writes to F1. Still, TableCache's design is relatively simple, especially when compared to alternatives such as write through caches: TableCache never needs to invalidate and reload an entire data set in response to writes. Furthermore, the write path at the database level can be unaware of the existence of the cache. This is critical, as putting cache updates on the critical path of F1 writes would re-introduce the tension we described above: Cache writes would have to be applied across shards and across geo-replicated regions, and to keep those fast, fewer and larger cache shards would be needed, in turn hurting reads.

7. PRODUCTION EXPERIENCE

Shasta powers multiple critical interactive reporting applications at Google. In this section, we report on our experience using Shasta for an important advertiser-facing application. Before Shasta, the views in this application were defined in C++ and computed using a custom query engine. When compared to this legacy system, Shasta has made software engineering more efficient, while also improving the performance and scalability of the system.

Software engineering efficiency: The design of the legacy system followed a common pattern for domain-specific backends, starting as a fairly simple C++ server and evolving into a mix of complex view definitions and query processing code. Without the strong boundaries that come with a declarative language like RVL, the query processing "engine" code overlapped with the code for view definitions, making it infeasible to separate ownership of the two pieces. As a result, feature development was bottlenecked on 15 engineers who had sole expert knowledge of both the custom engine and approximately 100 view definitions. Using Shasta's RVL view declarations, more than 200 code contributors across multiple teams now share view development, concentrating solely on business logic, while a smaller, separate team focuses on engine and compiler work. We replaced around 130k lines of C++ view definition code with 30k lines of RVL and 23k lines of C++ UDF code. The new code encapsulates all the logic of view transformations without relying on additional precomputed results for performance. This stateless design simplifies rollout of application changes and adds to the software engineering benefits of Shasta. For instance, changing a Shasta view column definition does not require updating data materializations previously stored using the old definition.

Relative system performance: The legacy reporting system was tailored for a particular storage schema, with limited support for query planning and distributed processing. Figures 7 and 8 show performance of Shasta relative to the legacy system for two views

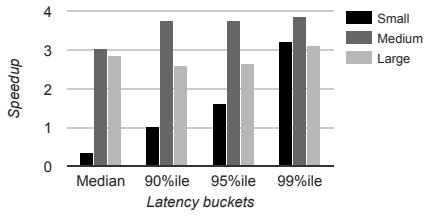

Figure 7: Speedup of Shasta vs. legacy system for View 1.

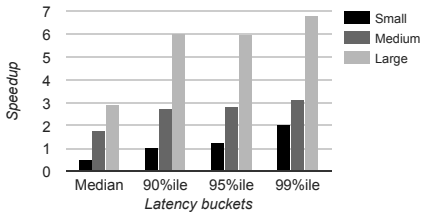

Figure 8: Speedup of Shasta vs. legacy system for View 2.

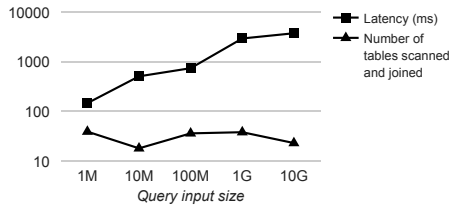

Figure 9: System scalability.

used in production. A "Speedup" value of 2 means Shasta is two times faster than the legacy system, 1 means latency is on par, and 0.5 means Shasta is two times slower. For each view, performance is compared for different query input data sizes (small, medium, large) which for this interactive workload are at most a few gigabytes per query. The views have varying business logic and query plan structures, yet the performance pattern is the same: Shasta is 2.5x to 7x faster for large queries and 2x to 4x faster for medium queries, thanks to more parallelism and data balancing via dynamic repartitioning available in F1's query engine. Only at the median for small queries does the legacy system consistently outperform Shasta. This is due to overhead inherent in the Shasta architecture, incurred by parsing and optimizing RVL code, generating SQL strings, parsing and planning SQL in F1 and scheduling query plans for execution. The overhead is typically modest in absolute terms (10s of ms), and not the majority portion of user-perceived latency, which includes network latency and latency from other systems higher in the stack. For larger and higher percentile queries on the other hand, the speedups achieved by Shasta were significant in terms of user-perceived latency. Future work may further reduce planning overhead using techniques such as prepared queries.

System scalability: While underlying tables in storage can be terabyte scale, Shasta queries issued by interactive applications are typically scoped to a single user's data. Carefully structured indexes in Mesa and data clustering by F1 root ID in TableCache then allow F1 queries to scan modest amounts of input – at most a few gigabytes for queries that run at interactive latencies. On the other hand, query plan complexity is high: important individual queries scan data from 50 tables and perform 60 join operations.

Despite the fact that input data size for most interactive queries is relatively modest, we made a conscious design decision to place highly distributed query processing at the core of the Shasta architecture, for two reasons. First, achieving reliably interactive latencies for workloads with the query complexities outlined above on single machines is challenging in Google's production environment, where commodity hardware shared between different services is the norm. Second, we find that in practice, sophisticated business applications are not truly limited to interactive queries with modest input sizes: As businesses evolve, data sizes and schema complexity tend to increase in ways that are hard to predict. Furthermore, applications tend to become *platforms*, offering to users not only interactive UIs, but also powerful programmatic developer APIs where query input sizes are larger and interactive latencies are less critical, yet where data freshness and consistency requirements are just as stringent. Shasta scales naturally to this setting and is used as a shared layer powering vastly different application interfaces of such platforms, guaranteeing fully consistent concepts and semantics in the different interfaces.

Combining data measured across one such platform at Google, Figure 9 illustrates the scalability of Shasta. As the size of input data increases, query latency has sublinear growth, made possible by F1's distributed query processing. Using statistical regression analysis, we found that the latency follows a trend of $O(n^{0.36})$ when n bytes are scanned. The chart also illustrates that structural query complexity – measured by the number of tables scanned and joined in the query – is largely constant across different input sizes. In Shasta applications, query input size tends to be determined by view parameters such as date ranges and the size of the given user's account, whereas query plan complexity is a function of schema complexity and application design, so the two are orthogonal.

Impact of TableCache: TableCache is critical for achieving low query latencies. The F1 query engine achieves at least 20x higher throughput reading from TableCache compared to reading from F1 storage directly. This is primarily due to the fact that TableCache partitions data much more finely, allowing much higher read parallelism, and leverages read-optimized data structures.

8. RELATED WORK

When comparing RVL to other query languages, one interesting feature to consider is implicit aggregation. The support for implicit aggregation has parallels to the MDX language [15] for querying OLAP data cubes. Dimensions and measures in the schema used by MDX are analogous to grouping columns and aggregatable columns in RVL, since measures are automatically aggregated for any selection of dimensions. RVL provides more flexibility than MDX, by supporting automatic aggregation on arbitrary joins. This makes RVL more readily applicable to existing SQL databases.

We can also compare RVL's view templates and sequential assignment statements to the features of other languages. The support for sequential subquery assignments has become particularly popular in query languages, as this syntax comes naturally to many developers. The scripting languages SCOPE [17] and Pig Latin [12] have support for assignment statements and output statements similar to RVL. The view templates of RVL allow blocks of assignment statements to be grouped together and reused with dynamic parameters, which can help developers organize a large code base. Spark SQL [2] supports the DataFrame API which can be used in various programming languages to formulate queries as a sequence of steps. Using DataFrames, a developer may create dynamic and reusable subqueries using constructs of the programming language (e.g., Java methods). RVL is intended to combine the best aspects of these approaches, allowing dynamic and reusable subqueries to be specified in one unified language.

TableCache acts as an extra layer of caching between the F1 query engine and F1 storage backed by Spanner. It can be compared to the usage of buffer pools in traditional database architectures [14, 6] since the goal is to exploit database access patterns to accelerate I/O. However, TableCache is read-only, which allows the design to be optimized for the read path without sacrificing write performance. In particular, TableCache uses fine-grained sharding to perform many small reads in parallel, and trans-

actional writes become more expensive with higher degrees of parallelism. Also, TableCache provides a higher-level abstraction than traditional block-based caches, where data from a table is accessed based on root ID and timestamp. Oracle TimesTen [10] can be deployed as a read-only cache in a similar way, but without timestamp-based access. We may also compare TableCache to the in-memory data management used by the scale-out extension (SOE) of SAP HANA [7], which allows reads at specific timestamps. The Shasta architecture differs from the SAP HANA SOE since the cache is decoupled from the query processing nodes, which allows TableCache to communicate directly with other server processes. Maintaining the freshness of TableCache is facilitated by F1 Change History which is conceptually based on the classical notion of write-ahead logging [11] in commercial DBMS engines. However, the implementation of F1 Change History is very different in that it is exposed as a first-class database entity which can be operated upon as a regular database table.

9. CONCLUDING REMARKS

Exposing interactive reporting and data analysis to users is crucial in business applications. For UIs to be truly interactive, user query latency must be low, and query results must provide fresh data that always include the most recent user updates. Data stores holding the transactional truth of underlying business data often have vast and complex schemas, placing significant transformation logic between stored data and user-facing concepts. The necessary data transformations often have to access data from multiple data stores, not all of which are read-optimized. In order to provide fresh data to applications while allowing for agile application development, it is ideal to avoid relying on precomputation and perform all transformations at query time. This results in two significant challenges: First, complex data transformations must be expressed in a way that scales to large engineering organizations and supports dynamic generation of online queries, based on rich query-time parameters. Second, the resulting online queries are complex – e.g., queries read from diverse data stores and join 50 or more tables – yet need to be executed reliably at interactive latencies.

Shasta takes an integrated approach to solving these challenges, using both language-level and system-level advances. In particular, Shasta provides a hybrid language interface to its users via RVL. RVL combines SQL-like functionality with limited but powerful procedural constructs. Using RVL, user queries can be stated succinctly and view definitions can naturally capture the dynamic nature of applications. At the system level, Shasta leverages a caching architecture that mitigates the impedance mismatch between stringent latency requirements for reads on the one hand and the underlying data store being mostly write-optimized on the other. Shasta also extends F1's distributed query processing framework to support user-defined function (UDF) calls at low latency yet with strong system isolation. As a result, Shasta incurs low latencies in computing complex views with little reliance on view materialization and without compromising on data freshness.

10. ACKNOWLEDGEMENTS

We would like to thank all current and past members of the Shasta team, especially Tuan Cao, Kelvin Lau, Patrick Lee, Wei-Hsin Lee, Curtis Menton, Aditi Pandit, Shashank Senapaty, Alejandro Valerio, and Junxiong Zhou. We'd also like to thank members of application teams that migrated to RVL, including Dmitry Churbanau, Daniel Halem, Raymond Ho, Anar Huseynov, Sujata Kosalge, Luke Snyder, and Andreas Sterbenz. We are grateful to the F1 team, including Brian Biskeborn, John Cieslewicz, Daniel Tenedorio, and Michael Styer, and to the Mesa team, including Mingsheng Hong, Kevin Lai, and Tao Zou. Finally, we thank Ashish Gupta, Sridhar Ramaswamy, and Jeff Shute for guidance throughout various stages of Shasta's development, and Nico Bruno, Ian Rae, Jeff Shute, and SIGMOD reviewers for insightful comments on our paper draft.

11. REFERENCES

[1] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, pages 577–588, 2013.

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

[3] 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, 2008.

[4] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3):8, 2013.

[6] W. Effelsberg and T. Haerder. Principles of database buffer management. *TODS*, 9(4):560–595, 1984.

[7] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *PVLDB*, 8(12):1716–1727, 2015.

[8] 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.

[9] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.

[10] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle TimesTen: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[11] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[12] 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.

[13] 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.

[14] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[15] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. *ACM SIGMOD Record*, 28(4):64–69, 1999.

[16] W. Yan and P.-A. Larson. Performing group-by before join. In *ICDE*, pages 89–100, 1994.

[17] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet MapReduce. *VLDB Journal*, 21(5):611–636, 2012.