Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources

Edmon Begoli
Oak Ridge National Laboratory (ORNL)
Oak Ridge, Tennessee, USA
begolie@ornl.gov

Jesús Camacho-Rodríguez
Hortonworks Inc.
Santa Clara, California, USA
jcamacho@hortonworks.com

Julian Hyde
Hortonworks Inc.
Santa Clara, California, USA
jhyde@hortonworks.com

Michael J. Mior
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
mmior@uwaterloo.ca

Daniel Lemire
University of Quebec (TELUQ)
Montreal, Quebec, Canada
lemire@gmail.com

ABSTRACT

Apache Calcite is a foundational software framework that provides query processing, optimization, and query language support to many popular open-source data processing systems such as Apache Hive, Apache Storm, Apache Flink, Druid, and MapD. Calcite's architecture consists of a modular and extensible query optimizer with hundreds of built-in optimization rules, a query processor capable of processing a variety of query languages, an adapter architecture designed for extensibility, and support for heterogeneous data models and stores (relational, semi-structured, streaming, and geospatial). This flexible, embeddable, and extensible architecture is what makes Calcite an attractive choice for adoption in big-data frameworks. It is an active project that continues to introduce support for the new types of data sources, query languages, and approaches to query processing and optimization.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**;

KEYWORDS

Apache Calcite, Relational Semantics, Data Management, Query Algebra, Modular Query Optimization, Storage Adapters

1 INTRODUCTION

Following the seminal System R, conventional relational database engines dominated the data processing landscape. Yet, as far back as 2005, Stonebraker and Çetintemel [49] predicted that we would see the rise a collection of specialized engines such as column stores, stream processing engines, text search engines, and so forth. They argued that specialized engines can offer more cost-effective performance and that they would bring the end of the "one size fits all" paradigm. Their vision seems today more relevant than ever. Indeed, many specialized open-source data systems have since become popular such as Storm [50] and Flink [16] (stream processing), Elasticsearch [15] (text search), Apache Spark [47], Druid [14], etc.

As organizations have invested in data processing systems tailored towards their specific needs, two overarching problems have arisen:

- The developers of such specialized systems have encountered related problems, such as query optimization [4, 25] or the need to support query languages such as SQL and related extensions (e.g., streaming queries [26]) as well as language-integrated queries inspired by LINQ [33]. Without a unifying framework, having multiple engineers independently develop similar optimization logic and language support wastes engineering effort.
- Programmers using these specialized systems often have to integrate several of them together. An organization might rely on Elasticsearch, Apache Spark, and Druid. We need to build systems capable of supporting optimized queries across heterogeneous data sources [55].

Apache Calcite was developed to solve these problems. It is a complete query processing system that provides much of the common functionality—query execution, optimization, and query languages—required by any database management system, except for data storage and management, which are left to specialized engines. Calcite was quickly adopted by Hive, Drill [13], Storm, and many other data processing engines, providing them with advanced query optimizations and query languages.[1] For example, Hive [24] is a popular data warehouse project built on top of Apache Hadoop. As Hive moved from its batch processing roots towards an interactive SQL query answering platform, it became clear that the project needed a powerful optimizer at its core. Thus, Hive adopted Calcite as its optimizer and their integration has been growing since. Many other projects and products have followed suit, including Flink, MapD [12], etc.

[1] http://calcite.apache.org/docs/powered_by

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190662

Furthermore, Calcite enables cross-platform optimization by exposing a common interface to multiple systems. To be efficient, the optimizer needs to reason globally, e.g., make decisions across different systems about materialized view selection.

Building a common framework does not come without challenges. In particular, the framework needs to be extensible and flexible enough to accommodate the different types of systems requiring integration.

We believe that the following features have contributed to Calcite's wide adoption in the open source community and industry:

- **Open source friendliness.** Many of the major data processing platforms of the last decade have been either open source or largely based on open source. Calcite is an open-source framework, backed by the Apache Software Foundation (ASF) [5], which provides the means to collaboratively develop the project. Furthermore, the software is written in Java making it easier to interoperate with many of the latest data processing systems [12, 13, 16, 24, 28, 44] that are often written themselves in Java (or in the JVM-based Scala), especially those in the Hadoop ecosystem.
- **Multiple data models.** Calcite provides support for query optimization and query languages using both streaming and conventional data processing paradigms. Calcite treats streams as time-ordered sets of records or events that are not persisted to the disk as they would be in conventional data processing systems.
- **Flexible query optimizer.** Each component of the optimizer is pluggable and extensible, ranging from rules to cost models. In addition, Calcite includes support for multiple planning engines. Hence, the optimization can be broken down into phases handled by different optimization engines depending on which one is best suited for the stage.
- **Cross-system support.** The Calcite framework can run and optimize queries across multiple query processing systems and database backends.
- **Reliability.** Calcite is reliable, as its wide adoption over many years has led to exhaustive testing of the platform. Calcite also contains an extensive test suite validating all components of the system including query optimizer rules and integration with backend data sources.
- **Support for SQL and its extensions.** Many systems do not provide their own query language, but rather prefer to rely on existing ones such as SQL. For those, Calcite provides support for ANSI standard SQL, as well as various SQL dialects and extensions, e.g., for expressing queries on streaming or nested data. In addition, Calcite includes a driver conforming to the standard Java API (JDBC).

The remainder is organized as follows. Section 2 discusses related work. Section 3 introduces Calcite's architecture and its main components. Section 4 describes the relational algebra at the core of Calcite. Section 5 presents Calcite's adapters, an abstraction to define how to read external data sources. In turn, Section 6 describes Calcite's optimizer and its main features, while Section 7 presents the extensions to handle different query processing paradigms. Section 8 provides an overview of the data processing systems already using Calcite. Section 9 discusses possible future extensions for the framework before we conclude in Section 10.

2 RELATED WORK

Though Calcite is currently the most widely adopted optimizer for big-data analytics in the Hadoop ecosystem, many of the ideas that lie behind it are not novel. For instance, the query optimizer builds on ideas from the Volcano [20] and Cascades [19] frameworks, incorporating other widely used optimization techniques such as materialized view rewriting [10, 18, 22]. There are other systems that try to fill a similar role to Calcite.

Orca [45] is a modular query optimizer used in data management products such as Greenplum and HAWQ. Orca decouples the optimizer from the query execution engine by implementing a framework for exchanging information between the two known as *Data eXchange Language*. Orca also provides tools for verifying the correctness and performance of generated query plans. In contrast to Orca, Calcite can be used as a standalone query execution engine that federates multiple storage and processing backends, including pluggable planners, and optimizers.

Spark SQL [3] extends Apache Spark to support SQL query execution which can also execute queries over multiple data sources as in Calcite. However, although the Catalyst optimizer in Spark SQL also attempts to minimize query execution cost, it lacks the dynamic programming approach used by Calcite and risks falling into local minima.

Algebricks [6] is a query compiler architecture that provides a data model agnostic algebraic layer and compiler framework for big data query processing. High-level languages are compiled to Algebricks logical algebra. Algebricks then generates an optimized job targeting the Hyracks parallel processing backend. While Calcite shares a modular approach with Algebricks, Calcite also includes a support for cost-based optimizations. In the current version of Calcite, the query optimizer architecture uses dynamic programming-based planning based on Volcano [20] with extensions for multi-stage optimizations as in Orca [45]. Though in principle Algebricks could support multiple processing backends (e.g., Apache Tez, Spark), Calcite has provided well-tested support for diverse backends for many years.

Garlic [7] is a heterogeneous data management system which represents data from multiple systems under a unified object model. However, Garlic does not support query optimization across different systems and relies on each system to optimize its own queries.

FORWARD [17] is a federated query processor that implements a superset of SQL called SQL++ [38]. SQL++ has a semi-structured data model that integrate both JSON and relational data models whereas Calcite supports semi-structured data models by representing them in the relational data model during query planning. FORWARD decomposes federated queries written in SQL++ into subqueries and executes them on the underlying databases according to the query plan. The merging of data happens inside the FORWARD engine.

Another federated data storage and processing system is BigDAWG, which abstracts a wide spectrum of data models including relational, time-series and streaming. The unit of abstraction in

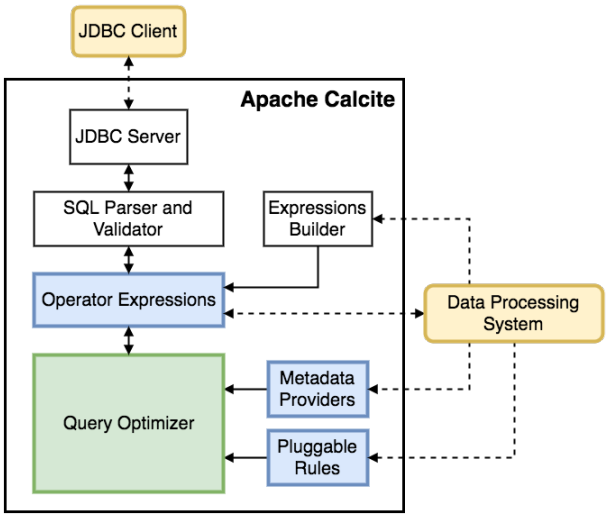


Figure 1: Apache Calcite architecture and interaction.

BigDAWG is called an *island of information*. Each island of information has a query language, data model and connects to one or more storage systems. Cross storage system querying is supported within the boundaries of a single island of information. Calcite instead provides a unifying relational abstraction which allows querying across backends with different data models.

Myria is a general-purpose engine for big data analytics, with advanced support for the Python language [21]. It produces query plans for other backend engines such as Spark and PostgreSQL.

3 ARCHITECTURE

Calcite contains many of the pieces that comprise a typical database management system. However, it omits some key components, e.g., storage of data, algorithms to process data, and a repository for storing metadata. These omissions are deliberate: it makes Calcite an excellent choice for mediating between applications having one or more data storage locations and using multiple data processing engines. It is also a solid foundation for building bespoke data processing systems.

Figure 1 outlines the main components of Calcite's architecture. Calcite's optimizer uses a tree of relational operators as its internal representation. The optimization engine primarily consists of three components: rules, metadata providers, and planner engines. We discuss these components in more detail in Section 6. In the figure, the dashed lines represent possible external interactions with the framework. There are different ways to interact with Calcite.

First, Calcite contains a query parser and validator that can translate a SQL query to a tree of relational operators. As Calcite does not contain a *storage layer*, it provides a mechanism to define table schemas and views in external storage engines via *adapters* (described in Section 5), so it can be used on top of these engines.

Second, although Calcite provides optimized SQL support to systems that need such database language support, it also provides optimization support to systems that already have their own language parsing and interpretation:

- Some systems support SQL queries, but without or with limited query optimization. For example, both Hive and Spark initially offered support for the SQL language, but they did not include an optimizer. For such cases, once the query has been optimized, Calcite can translate the relational expression back to SQL. This feature allows Calcite to work as a stand-alone system on top of any data management system with a SQL interface, but no optimizer.
- The Calcite architecture is not only tailored towards optimizing SQL queries. It is common that data processing systems choose to use their own parser for their own query language. Calcite can help optimize these queries as well. Indeed, Calcite also allows operator trees to be easily constructed by directly instantiating relational operators. One can use the built-in *relational expressions builder* interface. For instance, assume that we want to express the following Apache Pig [41] script using the expression builder:

```
emp = LOAD 'employee_data' AS (deptno, sal);
emp_by_dept = GROUP emp by (deptno);
emp_agg = FOREACH emp_by_dept GENERATE GROUP as deptno,
    COUNT(emp.sal) AS c, SUM(emp.sal) as s;
dump emp_agg;
```

The equivalent expression looks as follows:

```
final RelNode node = builder
  .scan("employee_data")
  .aggregate(builder.groupKey("deptno"),
             builder.count(false, "c"),
             builder.sum(false, "s", builder.field("sal")))
  .build();
```

This interface exposes the main constructs necessary for building relational expressions. After the optimization phase is finished, the application can retrieve the optimized relational expression which can then be mapped back to the system's query processing unit.

4 QUERY ALGEBRA

Operators. Relational algebra [11] lies at the core of Calcite. In addition to the operators that express the most common data manipulation operations, such as *filter*, *project*, *join* etc., Calcite includes additional operators that meet different purposes, e.g., being able to concisely represent complex operations, or recognize optimization opportunities more efficiently.

For instance, it has become common for OLAP, decision making, and streaming applications to use window definitions to express complex analytic functions such as moving average of a quantity over a time period or number or rows. Thus, Calcite introduces a *window* operator that encapsulates the window definition, i.e., upper and lower bound, partitioning etc., and the aggregate functions to execute on each window.

Traits. Calcite does not use different entities to represent logical and physical operators. Instead, it describes the physical properties associated with an operator using *traits*. These traits help the optimizer evaluate the cost of different alternative plans. Changing a trait value does not change the logical expression being evaluated, i.e., the rows produced by the given operator will still be the same.

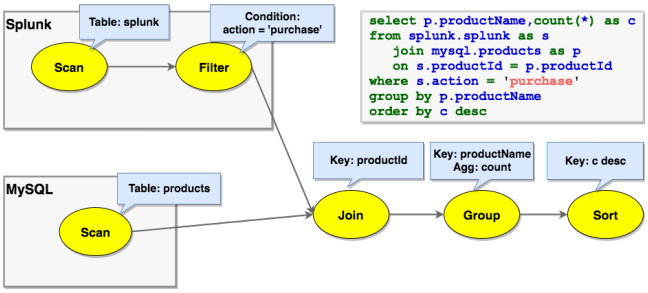


Figure 2: A Query Optimization Process.

During optimization, Calcite tries to enforce certain traits on relational expressions, e.g., the sort order of certain columns. Relational operators can implement a *converter* interface that indicates how to convert traits of an expression from one value to another.

Calcite includes common traits that describe the physical properties of the data produced by a relational expression, such as *ordering*, *grouping*, and *partitioning*. Similar to the SCOPE optimizer [57], the Calcite optimizer can reason about these properties and exploit them to find plans that avoid unnecessary operations. For example, if the input to the sort operator is already correctly ordered—possibly because this is the same order used for rows in the backend system—then the sort operation can be removed.

In addition to these properties, one of the main features of Calcite is the *calling convention* trait. Essentially, the trait represents the data processing system where the expression will be executed. Including the calling convention as a trait allows Calcite to meet its goal of optimizing transparently queries whose execution might span over different engines i.e., the convention will be treated as any other physical property.

For example, consider joining a *Products* table held in MySQL to an *Orders* table held in Splunk (see Figure 2). Initially, the scan of *Orders* takes place in the *splunk* convention and the scan of *Products* is in the *jdbc-mysql* convention. The tables have to be scanned inside their respective engines. The join is in the *logical* convention, meaning that no implementation has been chosen. Moreover, the SQL query in Figure 2 contains a filter (`where` clause) which is pushed into *splunk* by an adapter-specific rule (see Section 5). One possible implementation is to use Apache Spark as an external engine: the join is converted to *spark* convention, and its inputs are converters from *jdbc-mysql* and *splunk* to *spark* convention. But there is a more efficient implementation: exploiting the fact that Splunk can perform lookups into MySQL via ODBC, a planner rule pushes the join through the *splunk*-to-*spark* converter, and the join is now in *splunk* convention, running inside the Splunk engine.

5 ADAPTERS

An adapter is an architectural pattern that defines how Calcite incorporates diverse data sources for general access. Figure 3 depicts its components. Essentially, an adapter consists of a model, a schema, and a schema factory. The *model* is a specification of the physical properties of the data source being accessed. A *schema* is the definition of the data (format and layouts) found in the model. The data itself is physically accessed via tables. Calcite interfaces

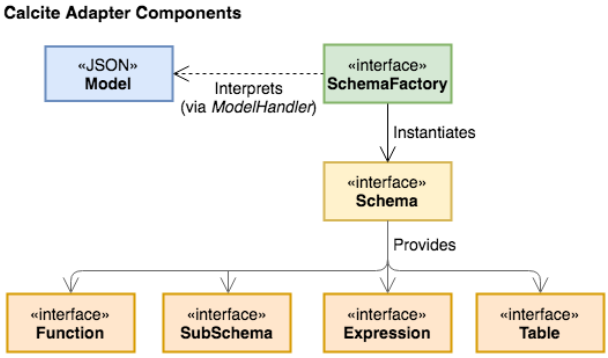


Figure 3: Calcite's Data Source Adapter Design.

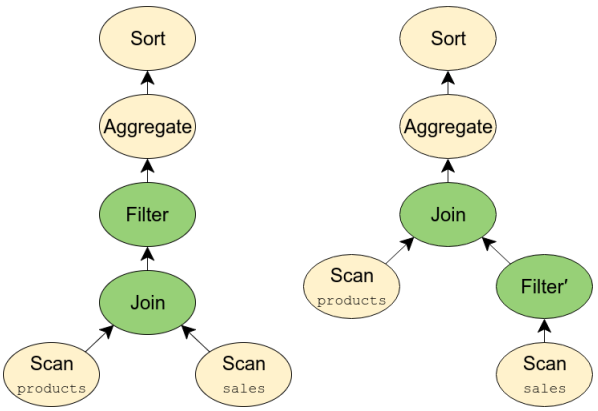
with the tables defined in the adapter to read the data as the query is being executed. The adapter may define a set of rules that are added to the planner. For instance, it typically includes rules to convert various types of logical relational expressions to the corresponding relational expressions of the adapter's convention. The *schema factory* component acquires the metadata information from the model and generates a schema.

As discussed in Section 4, Calcite uses a physical trait known as the *calling convention* to identify relational operators which correspond to a specific database backend. These physical operators implement the access paths for the underlying tables in each adapter. When a query is parsed and converted to a relational algebra expression, an operator is created for each table representing a scan of the data on that table. It is the minimal interface that an adapter must implement. If an adapter implements the table scan operator, the Calcite optimizer is then able to use client-side operators such as sorting, filtering, and joins to execute arbitrary SQL queries against these tables.

This table scan operator contains the necessary information the adapter requires to issue the scan to the adapter's backend database. To extend the functionality provided by adapters, Calcite defines an *enumerable* calling convention. Relational operators with the enumerable calling convention simply operate over tuples via an iterator interface. This calling convention allows Calcite to implement operators which may not be available in each adapter's backend. For example, the `EnumerableJoin` operator implements joins by collecting rows from its child nodes and joining on the desired attributes.

For queries which only touch a small subset of the data in a table, it is inefficient for Calcite to enumerate all tuples. Fortunately, the same rule-based optimizer can be used to implement adapter-specific rules for optimization. For example, suppose a query involves filtering and sorting on a table. An adapter which can perform filtering on the backend can implement a rule which matches a `LogicalFilter` and converts it to the adapter's calling convention. This rule converts the `LogicalFilter` into another `Filter` instance. This new `Filter` node has a lower associated cost that allows Calcite to optimize queries across adapters.

The use of adapters is a powerful abstraction that enables not only optimization of queries for a specific backend, but also across multiple backends. Calcite is able to answer queries involving tables across multiple backends by pushing down all possible logic to each



(a) Before **(b) After**

Figure 4: `FilterIntoJoinRule` application.

backend and then performing joins and aggregations on the resulting data. Implementing an adapter can be as simple as providing a table scan operator or it can involve the design of many advanced optimizations. Any expression represented in the relational algebra can be pushed down to adapters with optimizer rules.

6 QUERY PROCESSING AND OPTIMIZATION

The query optimizer is the main component in the framework. Calcite optimizes queries by repeatedly applying planner rules to a relational expression. A cost model guides the process, and the planner engine tries to generate an alternative expression that has the same semantics as the original but a lower cost.

Every component in the optimizer is extensible. Users can add relational operators, rules, cost models, and statistics.

Planner rules. Calcite includes a set of planner rules to transform expression trees. In particular, a rule matches a given pattern in the tree and executes a transformation that preserves semantics of that expression. Calcite includes several hundred optimization rules. However, it is rather common for data processing systems relying on Calcite for optimization to include their own rules to allow specific rewritings.

For example, Calcite provides an adapter for Apache Cassandra [29], a wide column store which partitions data by a subset of columns in a table and then within each partition, sorts rows based on another subset of columns. As discussed in Section 5, it is beneficial for adapters to push down as much query processing as possible to each backend for efficiency. A rule to push a `Sort` into Cassandra must check two conditions:

(1) the table has been previously filtered to a single partition (since rows are only sorted within a partition) and
(2) the sorting of partitions in Cassandra has some common prefix with the required sort.

This requires that a `LogicalFilter` has been rewritten to a `CassandraFilter` to ensure the partition filter is pushed down to the database. The effect of the rule is simple (convert a `LogicalSort` into a `CassandraSort`) but the flexibility in rule matching enables backends to push down operators even in complex scenarios.

For an example of a rule with more complex effects, consider the following query:

```
SELECT products.name, COUNT(*)
FROM sales JOIN products USING (productId)
WHERE sales.discount IS NOT NULL
GROUP BY products.name
ORDER BY COUNT(*) DESC;
```

The query corresponds to the relational algebra expression presented in Figure 4a. Because the `WHERE` clause only applies to the `sales` table, we can move the filter before the join as in Figure 4b. This optimization can significantly reduce query execution time since we do not need to perform the join for rows which do match the predicate. Furthermore, if the `sales` and `products` tables were contained in separate backends, moving the filter before the join also potentially enables an adapter to push the filter into the backend. Calcite implements this optimization via `FilterIntoJoinRule` which matches a filter node with a join node as a parent and checks if the filter can be performed by the join. This optimization illustrates the flexibility of the Calcite approach to optimization.

Metadata providers. Metadata is an important part of Calcite's optimizer, and it serves two main purposes: (*i*) guiding the planner towards the goal of reducing the cost of the overall query plan, and (*ii*) providing information to the rules while they are being applied.

Metadata providers are responsible for supplying that information to the optimizer. In particular, the default metadata providers implementation in Calcite contains functions that return the overall cost of executing a subexpression in the operator tree, the number of rows and the data size of the results of that expression, and the maximum degree of parallelism with which it can be executed. In turn, it can also provide information about the plan structure, e.g., filter conditions that are present below a certain tree node.

Calcite provides interfaces that allow data processing systems to plug their metadata information into the framework. These systems may choose to write providers that override the existing functions, or provide their own new metadata functions that might be used during the optimization phase. However, for many of them, it is sufficient to provide statistics about their input data, e.g., number of rows and size of a table, whether values for a given column are unique etc., and Calcite will do the rest of the work by using its default implementation.

As the metadata providers are pluggable, they are compiled and instantiated at runtime using Janino [27], a Java lightweight compiler. Their implementation includes a cache for metadata results, which yields significant performance improvements, e.g., when we need to compute multiple types of metadata such as *cardinality*, *average row size*, and *selectivity* for a given join, and all these computations rely on the cardinality of their inputs.

Planner engines. The main goal of a planner engine is to trigger the rules provided to the engine until it reaches a given objective. At the moment, Calcite provides two different engines. New engines are pluggable in the framework.

The first one, *a cost-based planner engine*, triggers the input rules with the goal of reducing the overall expression cost. The engine uses a dynamic programming algorithm, similar to Volcano [20], to create and track different alternative plans created by firing the

rules given to the engine. Initially, each expression is registered with the planner, together with a digest based on the expression attributes and its inputs. When a rule is fired on an expression e_1 and the rule produces a new expression e_2, the planner will add e_2 to the set of equivalence expressions S_a that e_1 belongs to. In addition, the planner generates a digest for the new expression, which is compared with those previously registered in the planner. If a similar digest associated with an expression e_3 that belongs to a set S_b is found, the planner has found a duplicate and hence will merge S_a and S_b into a new set of equivalences. The process continues until the planner reaches a configurable fix point. In particular, it can (*i*) exhaustively explore the search space until all rules have been applied on all expressions, or (*ii*) use a heuristic-based approach to stop the search when the plan cost has not improved by more than a given threshold δ in the last planner iterations. The cost function that allows the optimizer to decide which plan to choose is supplied through metadata providers. The default cost function implementation combines estimations for CPU, IO, and memory resources used by a given expression.

The second engine is an *exhaustive planner*, which triggers rules exhaustively until it generates an expression that is no longer modified by any rules. This planner is useful to quickly execute rules without taking into account the cost of each expression.

Users may choose to use one of the existing planner engines depending on their concrete needs, and switching from one to another, when their system requirements change, is straightforward. Alternatively, users may choose to generate *multi-stage optimization* logic, in which different sets of rules are applied in consecutive phases of the optimization process. Importantly, the existence of two planners allows Calcite users to reduce the overall optimization time by guiding the search for different query plans.

Materialized views. One of the most powerful techniques to accelerate query processing in data warehouses is the precomputation of relevant summaries or *materialized views*. Multiple Calcite adapters and projects relying on Calcite have their own notion of materialized views. For instance, Cassandra allows the user to define materialized views based on existing tables which are automatically maintained by the system.

These engines expose their materialized views to Calcite. The optimizer then has the opportunity to rewrite incoming queries to use these views instead of the original tables. In particular, Calcite provides an implementation of two different materialized view-based rewriting algorithms.

The first approach is based on *view substitution* [10, 18]. The aim is to substitute part of the relational algebra tree with an equivalent expression which makes use of a materialized view, and the algorithm proceeds as follows: (*i*) the scan operator over the materialized view and the materialized view definition plan are registered with the planner, and (*ii*) transformation rules that try to unify expressions in the plan are triggered. Views do not need to exactly match expressions in the query being replaced, as the rewriting algorithm in Calcite can produce partial rewritings that include additional operators to compute the desired expression, e.g., filters with residual predicate conditions.

The second approach is based on *lattices* [22]. Once the data sources are declared to form a lattice, Calcite represents each of the materializations as a *tile* which in turn can be used by the optimizer to answer incoming queries. On the one hand, the rewriting algorithm is especially efficient in matching expressions over data sources organized in a star schema, which are common in OLAP applications. On the other hand, it is more restrictive than view substitution, as it imposes restrictions on the underlying schema.

7 EXTENDING CALCITE

As we have mentioned in the previous sections, Calcite is not only tailored towards SQL processing. In fact, Calcite provides extensions to SQL expressing queries over other data abstractions, such as semi-structured, streaming and geospatial data. Its internal operators adapt to these queries. In addition to extensions to SQL, Calcite also includes a language-integrated query language. We describe these extensions throughout this section and provide some examples.

7.1 Semi-structured Data

Calcite supports several complex column data types that enable a hybrid of relational and semi-structured data to be stored in tables. Specifically, columns can be of type `ARRAY`, `MAP`, or `MULTISET`. Furthermore, these complex types can be nested so it is possible for example, to have a `MAP` where the values are of type `ARRAY`. Data within the `ARRAY` and `MAP` columns (and nested data therein) can be extracted using the `[]` operator. The specific type of values stored in any of these complex types need not be predefined.

For example, Calcite contains an adapter for MongoDB [36], a document store which stores documents consisting of data roughly equivalent to JSON documents. To expose MongoDB data to Calcite, a table is created for each document collection with a single column named `_MAP`: a map from document identifiers to their data. In many cases, documents can be expected to have a common structure. A collection of documents representing zip codes may each contain columns with a city name, latitude and longitude. It can be useful to expose this data as a relational table. In Calcite, this is achieved by creating a view after extracting the desired values and casting them to the appropriate type:

```
SELECT CAST(_MAP['city'] AS varchar(20)) AS city,
  CAST(_MAP['loc'][0] AS float) AS longitude,
  CAST(_MAP['loc'][1] AS float) AS latitude
FROM mongo_raw.zips;
```

With views over semi-structured data defined in this manner, it becomes easier to manipulate data from different semi-structured sources in tandem with relational data.

7.2 Streaming

Calcite provides first-class support for streaming queries [26] based on a set of streaming-specific extensions to standard SQL, namely *STREAM* extensions, windowing extensions, implicit references to streams via window expressions in joins, and others. These extensions were inspired by the Continuous Query Language [2] while also trying to integrate effectively with standard SQL. The primary extension, the `STREAM` directive tells the system that the user is interested in incoming records, not existing ones.

```
SELECT STREAM rowtime, productId, units
FROM Orders
WHERE units > 25;
```

In the absence of the STREAM keyword when querying a stream, the query becomes a regular relational query, indicating the system should process existing records which have already been received from a stream, not the incoming ones.

Due to the inherently unbounded nature of streams, windowing is used to unblock blocking operators such as aggregates and joins. Calcite's streaming extensions use SQL analytic functions to express sliding and cascading window aggregations, as shown in the following example.

```
SELECT STREAM rowtime,
  productId,
  units,
  SUM(units) OVER (ORDER BY rowtime
    PARTITION BY productId
    RANGE INTERVAL '1' HOUR PRECEDING) unitsLastHour
FROM Orders;
```

Tumbling, hopping and session windows[2] are enabled by the TUMBLE, HOPPING, SESSION functions and related utility functions such as TUMBLE_END and HOP_END that can be used respectively in GROUP BY clauses and projections.

```
SELECT STREAM
  TUMBLE_END(rowtime, INTERVAL '1' HOUR) AS rowtime,
  productId,
  COUNT(*) AS c,
  SUM(units) AS units
FROM Orders
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), productId;
```

Streaming queries involving window aggregates require the presence of monotonic or quasi-monotonic expressions in the GROUP BY clause or in the ORDER BY clause in case of sliding and cascading window queries.

Streaming queries which involve more complex stream-to-stream joins can be expressed using an implicit (time) window expression in the JOIN clause.

```
SELECT STREAM o.rowtime, o.productId, o.orderId,
  s.rowtime AS shipTime
FROM Orders AS o
JOIN Shipments AS s
  ON o.orderId = s.orderId
  AND s.rowtime BETWEEN o.rowtime AND
  o.rowtime + INTERVAL '1' HOUR;
```

In the case of an implicit window, Calcite's query planner validates that the expression is monotonic.

7.3 Geospatial Queries

Geospatial support is preliminary in Calcite, but is being implemented using Calcite's relational algebra. The core of this implementation consists in adding a new GEOMETRY data type which encapsulates different geometric objects such as points, curves, and polygons. It is expected that Calcite will be fully compliant with the OpenGIS Simple Feature Access [39] specification which defines a standard for SQL interfaces to access geospatial data. An example query finds the country which contains the city of Amsterdam:

```
SELECT name FROM (
  SELECT name,
    ST_GeomFromText('POLYGON((4.82 52.43, 4.97 52.43, 4.97 52.33,
      4.82 52.33, 4.82 52.43))') AS "Amsterdam",
    ST_GeomFromText(boundary) AS "Country"
  FROM country
) WHERE ST_Contains("Country", "Amsterdam");
```

[2]Tumbling, hopping, sliding, and session windows are different schemes for grouping of the streaming events [35].

7.4 Language-Integrated Query for Java

Calcite can be used to query multiple data sources, beyond just relational databases. But it also aims to support more than just the SQL language. Though SQL remains the primary database language, many programmers favour language-integrated languages like LINQ [33]. Unlike SQL embedded within Java or C++ code, language-integrated query languages allow the programmer to write all of her code using a single language. Calcite provides Language-Integrated Query for Java (or LINQ4J, in short) which closely follows the convention set forth by Microsoft's LINQ for the .NET languages.

8 INDUSTRY AND ACADEMIA ADOPTION

Calcite enjoys wide adoption, specially among open-source projects used in industry. As Calcite provides certain integration flexibility, these projects have chosen to either (*i*) embed Calcite within their core, i.e., use it as a library, or (*ii*) implement an adapter to allow Calcite to federate query processing. In addition, we see a growing interest in the research community to use Calcite as the cornerstone of the development of data management projects. In the following, we describe how different systems are using Calcite.

8.1 Embedded Calcite

Table 1 provides a list of software that incorporates Calcite as a library, including (*i*) the query language interface that they expose to users, (*ii*) whether they use Calcite's JDBC driver (called *Avatica*), (*iii*) whether they use the SQL parser and validator included in Calcite, (*iv*) whether they use Calcite's query algebra to represent their operations over data, and (*v*) the engine that they rely on for execution, e.g., their own native engine, Calcite's operators (referred to as *enumerable*), or any other project.

Drill [13] is a flexible data processing engine based on the Dremel system [34] that internally uses a schema-free JSON document data model. Drill uses its own dialect of SQL that includes extensions to express queries on semi-structured data, similar to SQL++ [38].

Hive [24] first became popular as a SQL interface on top of the MapReduce programming model [52]. It has since moved towards being an interactive SQL query answering engine, adopting Calcite as its rule and cost-based optimizer. Instead of relying on Calcite's JDBC driver, SQL parser and validator, Hive uses its own implementation of these components. The query is then translated into Calcite operators, which after optimization are translated into Hive's physical algebra. Hive operators can be executed by multiple engines, the most popular being Apache Tez [43, 51] and Apache Spark [47, 56].

Apache Solr [46] is a popular full-text distributed search platform built on top of the Apache Lucene library [31]. Solr exposes multiple query interfaces to users, including REST-like HTTP/XML and JSON APIs. In addition, Solr integrates with Calcite to provide SQL compatibility.

Apache Phoenix [40] and Apache Kylin [28] both work on top of Apache HBase [23], a distributed key-value store modeled after Bigtable [9]. In particular, Phoenix provides a SQL interface and orchestration layer to query HBase. Kylin focuses on OLAP-style

System	Query Language	JDBC Driver	SQL Parser and Validator	Relational Algebra	Execution Engine
Apache Drill	SQL + extensions	✓	✓	✓	Native
Apache Hive	SQL + extensions			✓	Apache Tez, Apache Spark
Apache Solr	SQL	✓	✓	✓	Native, Enumerable, Apache Lucene
Apache Phoenix	SQL	✓	✓	✓	Apache HBase
Apache Kylin	SQL	✓	✓		Enumerable, Apache HBase
Apache Apex	Streaming SQL	✓	✓	✓	Native
Apache Flink	Streaming SQL	✓	✓	✓	Native
Apache Samza	Streaming SQL	✓	✓	✓	Native
Apache Storm	Streaming SQL	✓	✓	✓	Native
MapD [32]	SQL		✓	✓	Native
Lingual [30]	SQL		✓	✓	Cascading
Qubole Quark [42]	SQL	✓	✓	✓	Apache Hive, Presto

Table 1: List of systems that embed Calcite.

Adapter	Target language
Apache Cassandra	Cassandra Query Language (CQL)
Apache Pig	Pig Latin
Apache Spark	Java (Resilient Distributed Datasets)
Druid	JSON
Elasticsearch	JSON
JDBC	SQL (multiple dialects)
MongoDB	Java
Splunk	SPL

Table 2: List of Calcite adapters.

SQL queries instead, building cubes that are declared as materialized views and stored in HBase, and hence allowing Calcite's optimizer to rewrite the input queries to be answered using those cubes. In Kylin, query plans are executed using a combination of Calcite native operators and HBase.

Recently Calcite has become popular among streaming systems too. Projects such as Apache Apex [1], Flink [16], Apache Samza [44], and Storm [50] have chosen to integrate with Calcite, using its components to provide a streaming SQL interface to their users. Finally, other commercial systems have adopted Calcite, such as MapD [32], Lingual [30], and Qubole Quark [42].

8.2 Calcite Adapters

Instead of using Calcite as a library, other systems integrate with Calcite via adapters which read their data sources. Table 2 provides the list of available adapters in Calcite. One of the main key components of the implementation of these adapters is the *converter* responsible for translating the algebra expression to be pushed to the system into the query language supported by that system. Table 2 also shows the languages that Calcite translates into for each of these adapters.

The JDBC adapter supports the generation of multiple SQL dialects, including those supported by popular RDBMSes such as PostgreSQL and MySQL. In turn, the adapter for Cassandra [8] generates its own SQL-like language called *CQL* whereas the adapter for Apache Pig [41] generates queries expressed in Pig Latin [37]. The adapter for Apache Spark [47] uses the Java RDD API. Finally, Druid [14], Elasticsearch [15] and Splunk [48] are queried through REST HTTP API requests. The queries generated by Calcite for these systems are expressed in JSON or XML.

8.3 Uses in Research

In a research setting, Calcite has been considered [54] as a polystore-alternative for precision medicine and clinical analysis scenarios. In those scenarios, heterogeneous medical data has to be logically assembled and aligned to assess the best treatments based on the comprehensive medical history and the genomic profile of the patient. The data comes from relational sources representing patients' electronic medical records, structured and semi-structured sources representing various reports (oncology, psychiatry,laboratory tests, radiology, etc.), imaging, signals, and sequence data, stored in scientific databases. In those circumstances, Calcite represents a good foundation with its uniform query interface, and flexible adapter architecture, but the ongoing research efforts are aimed at (*i*) introduction of the new adapters for array, and textual sources, and (*ii*) support efficient joining of heterogeneous data sources.

9 FUTURE WORK

The future work on Calcite will focus on the development of the new features, and the expansion of its adapter architecture:

- Enhancements to the design of Calcite to further support its use a standalone engine, which would require a support for data definition languages (DDL), materialized views, indexes and constraints.
- Ongoing improvements to the design and flexibility of the planner, including making it more modular, allowing users Calcite to supply planner *programs* (collections of rules organized into planning phases) for execution.
- Incorporation of new parametric approaches [53] into the design of the optimizer.
- Support for an extended set of SQL commands, functions, and utilities, including full compliance with OpenGIS.
- New adapters for non-relational data sources such as array databases for scientific computing.
- Improvements to performance profiling and instrumentation.

9.1 Performance Testing and Evaluation

Though Calcite contains a performance testing module, it does not evaluate query execution. It would be useful to assess the performance of systems built with Calcite. For example, we could compare the performance of Calcite with similar frameworks. Unfortunately, it might be difficult to craft fair comparisons. For example, like Calcite, Algebricks optimizes queries for Hive. Borkar et al. [6] compared Algebricks with the Hyracks scheduler against Hive version 0.12 (without Calcite). The work of Borkar et al. precedes significant engineering and architectural changes into Hive. Comparing Calcite against Algebricks in a fair manner in terms of timings does not seem feasible, as one would need to ensure that each uses the same execution engine. Hive applications rely mostly on either Apache Tez or Apache Spark as execution engines whereas Algebricks is tied to its own framework (including Hyracks).

Moreover, to assess the performance of Calcite-based systems, we need to consider two distinct use cases. Indeed, Calcite can be used either as part of a single system—as a tool to accelerate the construction of such a system—or for the more difficult task of combining several distinct systems—as a common layer. The former is tied to the characteristics of the data processing system, and because Calcite is so versatile and widely used, many distinct benchmarks are needed. The latter is limited by the availability of existing heterogeneous benchmarks. BigDAWG [55] has been used to integrate PostgreSQL with Vertica, and on a standard benchmark, one gets that the integrated system is superior to a baseline where entire tables are copied from one system to another to answer specific queries. Based on real-world experience, we believe that more ambitious goals are possible for integrated multiple systems: they should be superior to the sum of their parts.

10 CONCLUSION

Emerging data management practices and associated analytic uses of data continue to evolve towards an increasingly diverse, and heterogeneous spectrum of scenarios. At the same time, relational data sources, accessed through the SQL, remain an essential means to how enterprises work with the data. In this somewhat dichotomous space, Calcite plays a unique role with its strong support for both traditional, conventional data processing, and for its support of other data sources including those with semi-structured, streaming and geospatial models. In addition, Calcite's design philosophy with a focus on flexibility, adaptivity, and extensibility, has been another factor in Calcite becoming the most widely adopted query optimizer, used in a large number of open-source frameworks. Calcite's dynamic and flexible query optimizer, and its adapter architecture allows it to be embedded selectively by a variety of data management frameworks such as Hive, Drill, MapD, and Flink. Calcite's support for heterogeneous data processing, as well as for the extended set of relational functions will continue to improve, in both functionality and performance.

ACKNOWLEDGMENTS

We would like to thank the Calcite community, contributors and users, who build, maintain, use, test, write about, and continue to push the Calcite project forward. This manuscript has been in part co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

REFERENCES

[1] Apex. Apache Apex. https://apex.apache.org. (Nov. 2017).
[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. *The CQL Continuous Query Language: Semantic Foundations and Query Execution.* Technical Report 2003-67. Stanford InfoLab.
[3] Michael Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394.
[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394.
[5] ASF. The Apache Software Foundation. (Nov. 2017). Retrieved November 20, 2017 from http://www.apache.org/
[6] Vinayak Borkar, Yingyi Bu, E. Preston Carman, Jr., Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J. Carey, and Vassilis J. Tsotras. 2015. Algebricks: A Data Model-agnostic Compiler Backend for Big Data Languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 422–433.
[7] M. J. Carey et al. 1995. Towards heterogeneous multimedia information systems: the Garlic approach. In *IDE-DOM '95*. 124–131.
[8] Cassandra. Apache Cassandra. (Nov. 2017). Retrieved November 20, 2017 from http://cassandra.apache.org/
[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 205–218.
[10] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE '95)*. IEEE Computer Society, Washington, DC, USA, 190–200.
[11] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387.
[12] Alex Şuhan. Fast and Flexible Query Analysis at MapD with Apache Calcite. (feb 2017). Retrieved November 20, 2017 from https://www.mapd.com/blog/2017/02/08/fast-and-flexible-query-analysis-at-mapd-with-apache-calcite-2/
[13] Drill. Apache Drill. (Nov. 2017). Retrieved November 20, 2017 from http://drill.apache.org/
[14] Druid. Druid. (Nov. 2017). Retrieved November 20, 2017 from http://druid.io/
[15] Elastic. Elasticsearch. (Nov. 2017). Retrieved November 20, 2017 from https://www.elastic.co
[16] Flink. Apache Flink. https://flink.apache.org. (Nov. 2017).
[17] Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Michalis Petropoulos. 2011. The SQL-based all-declarative FORWARD web application development framework. In *CIDR*.

[18] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *SIGMOD Rec.* 30, 2 (May 2001), 331–342.

[19] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* (1995).

[20] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 209–218.

[21] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 881–884.

[22] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. *SIGMOD Rec.* 25, 2 (June 1996), 205–216.

[23] HBase. Apache HBase. (Nov. 2017). Retrieved November 20, 2017 from http://hbase.apache.org/

[24] Hive. Apache Hive. (Nov. 2017). Retrieved November 20, 2017 from http://hive.apache.org/

[25] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1235–1246.

[26] Julian Hyde. 2010. Data in Flight. *Commun. ACM* 53, 1 (Jan. 2010), 48–52.

[27] Janino. Janino: A super-small, super-fast Java compiler. (Nov. 2017). Retrieved November 20, 2017 from http://www.janino.net/

[28] Kylin. Apache Kylin. (Nov. 2017). Retrieved November 20, 2017 from http://kylin.apache.org/

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

[30] Lingual. Lingual. (Nov. 2017). Retrieved November 20, 2017 from http://www.cascading.org/projects/lingual/

[31] Lucene. Apache Lucene. (Nov. 2017). Retrieved November 20, 2017 from https://lucene.apache.org/

[32] MapD. MapD. (Nov. 2017). Retrieved November 20, 2017 from https://www.mapd.com

[33] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 706–706.

[34] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339. http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R29.pdf

[35] Marcelo RN Mendes, Pedro Bizarro, and Paulo Marques. 2009. A performance study of event processing systems. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 221–236.

[36] Mongo. MongoDB. (Nov. 2017). Retrieved November 28, 2017 from https://www.mongodb.com/

[37] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*.

[38] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631* (2014).

[39] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option. http://portal.opengeospatial.org/files/?artifact_id=25355. (2010).

[40] Phoenix. Apache Phoenix. (Nov. 2017). Retrieved November 20, 2017 from http://phoenix.apache.org/

[41] Pig. Apache Pig. (Nov. 2017). Retrieved November 20, 2017 from http://pig.apache.org/

[42] Qubole Quark. Qubole Quark. (Nov. 2017). Retrieved November 20, 2017 from https://github.com/qubole/quark

[43] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1357–1369. https://doi.org/10.1145/2723372.2742790

[44] Samza. Apache Samza. (Nov. 2017). Retrieved November 20, 2017 from http://samza.apache.org/

[45] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 337–348.

[46] Solr. Apache Solr. (Nov. 2017). Retrieved November 20, 2017 from http://lucene.apache.org/solr/

[47] Spark. Apache Spark. (Nov. 2017). Retrieved November 20, 2017 from http://spark.apache.org/

[48] Splunk. Splunk. (Nov. 2017). Retrieved November 20, 2017 from https://www.splunk.com/

[49] Michael Stonebraker and Ugur Çetintemel. 2005. "One size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE Computer Society, Washington, DC, USA, 2–11.

[50] Storm. Apache Storm. (Nov. 2017). Retrieved November 20, 2017 from http://storm.apache.org/

[51] Tez. Apache Tez. (Nov. 2017). Retrieved November 20, 2017 from http://tez.apache.org/

[52] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *VLDB* (2009), 1626–1629.

[53] Immanuel Trummer and Christoph Koch. 2017. Multi-objective parametric query optimization. *The VLDB Journal* 26, 1 (2017), 107–124.

[54] Ashwin Kumar Vajantri, Kunwar Deep Singh Toor, and Edmon Begoli. 2017. An Apache Calcite-based Polystore Variation for Federated Querying of Heterogeneous Healthcare Sources. In *2nd Workshop on Methods to Manage Heterogeneous Big Data and Polystore Databases*. IEEE Computer Society, Washington, DC, USA.

[55] Katherine Yu, Vijay Gadepally, and Michael Stonebraker. 2017. Database engine integration and performance analysis of the BigDAWG polystore system. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE Computer Society, Washington, DC, USA, 1–7.

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.

[57] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE Computer Society, Washington, DC, USA, 1060–1071.