SUDAF: Sharing User-Defined Aggregate Functions

Chao Zhang, Farouk Toumani and Bastien Doreau
LIMOS, CNRS, University Clermont Auvergne
zhangchaohit13sg@gmail.com, ftoumani@isima.fr, bastien.doreau@isima.fr

***Abstract*—We present SUDAF (Sharing User-Defined Aggregate Functions), a declarative framework that allows users to formulate UDAFs as mathematical expressions and use them in SQL statements. SUDAF rewrites partial aggregates of UDAFs using built-in aggregate functions and supports efficient dynamic caching and reusing of partial aggregates. Our evaluation shows that using SUDAF on top of Spark SQL can lead from one to two orders of magnitude improvement in query execution times compared to the original Spark SQL.**

***Index Terms*—Query processing, Query Rewriting, User-defined aggregate functions.**

I. Introduction

Modern data systems enable users to extend the system functionalities by defining their aggregations and use them in SQL statements. The UDAF (User-Defined Aggregate Function) mechanism provides a flexible interface to allow users to define new aggregate functions that can then be used for advanced data analytics, e.g., queries with statistical functions or ML workloads. Current UDAF mechanisms suffer, however, from at least two drawbacks. Firstly, defining a UDAF is not an easy task since it is up to the user to implement the routine that computes the aggregation function. For example, to write a custom UDAF in Spark SQL [1], a user needs to map the UDAF to four methods: initialize, update, merge and evaluate. The user must ensure that the merge method is commutative and associative such that the UDAF can be computed correctly in a spark cluster. Secondly, the semantics of a hardcoded UDAF, i.e., what properties it has, may not be totally known by the query processor, which hampers the optimization possibilities.

To overcome the above issues, we design SUDAF, a framework that comes equipped with the two novel functionalities: (i) a declarative framework that allows users to write UDAFs as mathematical expressions and use them in SQL queries[1], and (ii) a dynamic approach for rewriting partial aggregates of UDAFs and caching their computation results to optimize the computations of future UDAFs on the fly.

In order to provide the above two functionalities, SUDAF uses a canonical form of aggregations as an internal representation of UDAFs. An aggregation can be expressed as a triple $(F, \oplus, T)$ [7], where F is a translating function, $\oplus$ is a commutative and associative binary operation and T is a terminating function. We present canonical forms of the geometric mean, the skewness and the covariance in table I. SUDAF decomposes a UDAF based on the canonical form to obtain the following two parts of a UDAF: (1) partial aggregates, the F and $\oplus$ components in $(F, \oplus, T)$, which is usually expensive to compute since it requires scanning base table and applying hash or sort aggregate; and (2) a terminating function, the T component in $(F, \oplus, T)$, which is the final computation of a UDAF taking the results of partial aggregates as inputs. This separation between partial aggregates and a terminating function allows SUDAF to focus on optimizing the computation of the expensive part, partial aggregates, and to release a terminating function to be arbitrary functions.

TABLE I
Examples of aggregations in canonical forms.

Aggregation	Expression	Canonical form $\left(\mathbf{F}, \oplus, \mathbf{T}\right)$
Geometric mean	$(\prod x_i)^{1/n}$	$\left((x_i, 1), (\times, +), (s_1)^{1/s_2}\right)$
Skewness	$\frac{(\sum(x_i - avg)^3)/n}{((\sum(x_i - avg)^2)/n)^{3/2}}$	$\left(((x_i - avg)^3, (x_i - avg)^2, 1), (+, +, +), \frac{(s_1/s_3)}{(s_2/s_3)^{3/2}}\right)$
Covariance	$\frac{\sum(x_i y_i)}{n} - \frac{\sum x_i \sum y_i}{n^2}$	$\left((x_i, y_i, x_i y_i, 1), (+, +, +, +), \frac{s_3}{s_4} - \frac{s_1 s_2}{s_4}\right)$

SUDAF builds on the above canonical form of aggregations to provide two optimization mechanisms: (1) rewriting partial aggregates using built-in functions, and (2) identifying appropriate partial aggregates that can be cached and reused to compute new ones. The first approach opens the black boxes of UDAFs and brings substantial benefits for query engines. For example, Spark SQL includes the rewritten built-in functions in its *WholeStageCodeGen* [2], but it does not do that for a UDAF. The second optimization can significantly speed up query execution time since the expensive computations, i.e., scanning and hash aggregate, can be skipped. SUDAF accomplish this sharing functionality by analyzing the semantics of partial aggregates, i.e., their mathematical expressions.

Our online technical report [6] contains additional materials: (i) the theoretical foundations of SUDAF, (ii) empirical comparisons with Spark SQL [1] and PostgreSQL [9], and (iii) extensive performance evaluation.

Illustrating examples: We illustrate SUDAF's functionalities using the following examples. We consider the following 4 relations store_sales, store, date_dim and stores. Consider a hypothesis of a *simple linear regression*: $y = \theta_1 x + \theta_0$, where $\theta_0(X, Y) = avg(Y) - \theta_1 avg(X)$ and $\theta_1(X, Y) =$

[1]This approach is more intuitive than hard-coding UDAFs, e.g., Wolfram Mathematica offers math expressions for defining statistical computation.

[2]This mechanism enables to fuse several physical operators into a single function, such that virtual function calls can be reduced, which improves query running time for in-memory computation

$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$. Suppose that a user want to compute θ_1 and θ_0 in the following query Q1. θ_1 can be hardcoded as a user-defined function, called theta1(), and then it can be used in an SQL statement, e.g., writing Scala code in Spark SQL.

```
Q1: SELECT item_sk, year, avg(list_price),
           avg(sales_price),
           theta1(list_price,sales_price)
    FROM   store_sales, store, date_dim
    WHERE  sold_date_sk = date_sk and state = 'TN' and
           ss_store_sk = s_store_sk
    GROUP BY item_sk, year;
```

In SUDAF, theta1() can be defined declaratively using its expression without needs of any programming effort.

The first step to process Q1 in SUDAF is to factor out partial aggregates of theta1() and avg() and to rewrite them using built-in functions. More precisely, SUDAF identifies the following 5 partial aggregates in the expression of θ_1: $s_1 = count(), s_2 = \sum x_i, s_3 = \sum x_i^2, s_4 = \sum y_i$ and $s_5 = \sum x_i y_i$. Hence, SUDAF rewrites Q1 to the following query RQ1 where the partial aggregates are first computed and then theta1() is computed using the partial aggregates.

```
RQ1: SELECT item_sk, year, s2/s1 avg_lp, s4/s1 avg_sp,
     (s1*s5-s4*s2)/NULLIF((s1*s3-power(s2,2)),0) theta1
      FROM   (SELECT item_sk, year, count(*) s1,
                     sum(list_price) s2,
                     sum(power(list_price,2)) s3,
                     sum(sales_price) s4,
                     sum(sales_price*list_price) s5
              FROM   store_sales, store, date_dim
              WHERE  sold_date_sk = date_sk and
                     ss_store_sk = s_store_sk and
                     state = 'TN'
              GROUP BY item_sk, year) TEMP;
```

Compared to the query Q1, RQ1 uses only built-in aggregate functions and hence it is expected to be much more efficient. Moreover, SUDAF caches the partial aggregates of RQ1, which are $s1, s2, s3, s4$ and $s5$, to optimize future queries with UDAFs.

Assume that the user issues a second query Q2 that computes quadratic mean qm() and standard deviation stddev():

```
Q2: SELECT item_sk, year, qm(list_price),
           stddev(list_price)
    FROM   store_sales, store, date_dim
    WHERE  sold_date_sk = date_sk and
           ss_store_sk = s_store_sk and state = 'TN'
    GROUP BY item_sk, year;
```

Similarly, qm() and stddev() can be defined using their expressions in SUDAF. When processing Q2, SUDAF factors out their partial aggregations and generates the following query RQ2 having partial aggregates s_1, s_2 and s_3.

```
RQ2: SELECT item_sk, year, sqrt(s3/s1) qm_lp,
            sqrt(s3/s1-power(s2/s1,2)) std_lp
      FROM   (SELECT item_sk, year, count(*) s1,
                     sum(list_price) s2,
                     sum(power(list_price,2)) s3
              FROM   store_sales, store, date_dim
              WHERE  sold_date_sk = date_sk and
                     ss_store_sk = s_store_sk and
                     state = 'TN'
              GROUP BY item_sk, year) TEMP2;
```

SUDAF automatically identifies the opportunity to reuse the cached partial aggregates of RQ1 for computing aggregates of RQ2. This makes the processing of Q2 significantly faster than computing the query Q2 from scratch.

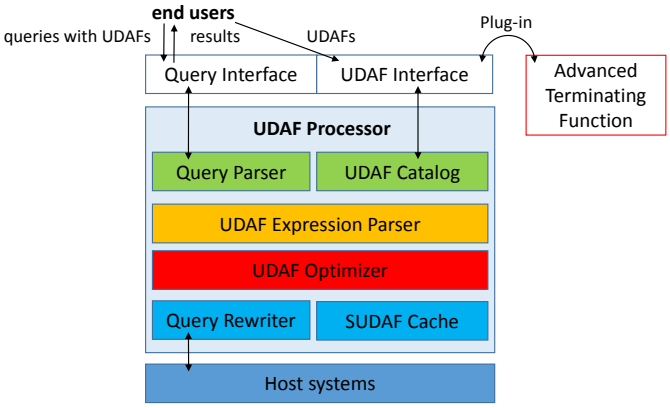

Fig. 1. SUDAF architecture.

The remainder of this paper is organized as follows. We present the system architecture of SUDAF in section II, and we report on the performance evaluation of SUDAF in section III. Our demonstration scenarios, described in section IV, allow users to define UDAFs using their mathematical expressions, to execute a workload of queries containing defined UDAFs and visualize their computation results and execution times.

II. System Overview

SUDAF adopts a three-tier architecture, UDAF interface, UDAF processor and a host system, which are presented in figure 1. We present each of them below.

UDAF interface: The top layer is the UDAF interface. As depicted in figure 1, end users can declare mathematical expressions of aggregations at the UDAF interface and use them in SQL queries at the query interface. Advanced terminating function interface is put aside to create a function that is not possible to be expressed using simple arithmetical operators supported in SUDAF, e.g., the moment solver used to estimate quantile [2]. In this context, one can define partial aggregates and use them as inputs of a hardcoded function to create a UDAF, i.e., approximating median using moment sketch [2].

UDAF processor: The key components of the UDAF processor shown in figure 1 are detailed below.

(a) UDAF Catalog. SUDAF requires users to declare the mathematical expression of a UDAF and provide a name for the expression, which will be registered in UDAF Catalog.

(b) Query Parser. We rely on JSqlParser [3] to parse a SQL statement. Query parser identifies the UDAFs and aggregate columns (columns that UDAFs are applied on) in a query.

(c) UDAF Expression Parser. This component is in charge of translating the mathematical expressions of aggregations to its canonical forms represented as an AET (aggregate expression tree). An AET can be seen as a logical plan for computing an aggregation.

(d) SUDAF Cache. SUDAF identifies appropriate partial aggregates of UDAFs and caches their computation results

to reuse for computing future UDAFs. SUDAF rests on the canonical form of aggregations to identify fine-grained partial aggregates, which maximizes the sharing opportunities. SUDAF cache is sophisticatedly organized based on the expressions of fine-grained partial aggregates. The caches can be accessed by a symbolic index, which maps a partial aggregate to a cached value that can be used to compute the partial aggregate.

(e) UDAF Optimizer. The first functionality of the UDAF optimizer is to apply rule-based optimizations on AETs to obtain fine-grained partial aggregates. Moreover, the UDAF optimizer identifies sharing possibilities w.r.t. the content of caches by computing entries of the symbolic index. The symbolic index is the bridge between SUDAF cache and UDAF optimizer. The UDAF optimizer will also compute the terminating function of a UDAF using caches to obtain the final results of a UDAF.

(f) Query Rewriter. In the case that SUDAF cache cannot be reused to compute partial aggregations, query rewriter rewrites received queries with UDAFs to one with built-in functions.

Host system: SUDAF sends SQL queries with built-in functions to host systems (any SQL engines). In our evaluation and demonstration, we will use Spark SQL as the host system. Our online technical report [6] includes comprehensive experimental evaluations using PostgreSQL [9] and Spark SQL [1] separately as a host system.

III. Evaluation

This section presents the runtimes of computing workloads of queries using SUDAF and Spark SQL. All experiments are performed on a Spark cluster with one master node and six worker nodes, running ubuntu server 16.04, Spark 2.2.0 and Hadoop 2.7.4. The master node has a processor of 6 cores (XEON E5-2630 2.4GHz), 16 GB of main memory and 160 GB of disk space, and every worker node has a processor of 4 cores (XEON E5-2630 2.4GHz), 8 GB of main memory and 80 GB of disk space. All experiments are evaluated on the Milan Telecommunication dataset [5] in Parquet format, 319 million rows in total.

We use the following query models in our experiments, where AGG represents an aggregation.

```
SELECT square_id, AGG(internet_traffic) FROM milan_TC
GROUP by square_id ORDER by square_id LIMIT 20;
```

We use the following 11 aggregate functions to instantiate the query model: cubic_mean (cm), quadratic_mean (qm), geometric_mean (gm), harmonic_mean (hm), min, max, count, sum, average (avg), standard deviation (std), variance (var). We simulate the 11 instances coming in 2 orders shown below,

AS1 = [cm, qm, gm, hm, min, max, count, std, var, sum, avg]

AS2 = [max, min, sum, avg, count, std, var, cm, gm, hm, qm]

Thus, we have two query sequences, which follows the order AS1 and AS2. In the used Spark SQL version, all of these functions are built-in except cm, qm, gm and hm, which are implemented using *UserDefinedAggregateFunction* in Scala. In the sequence AS2, we prefetch a moment sketch (MS), which can be used to approximate a percentile, e.g., median.

Our first observation is that SUDAF speeds up UDAF queries 2X (the cases cm, qm, gm and hm shown in figure 2 (e) and (f). The major reason for this improvement is that SUDAF rewrites queries with UDAFs to queries with partial aggregations that can be evaluated using Spark SQL built-in functions, which are faster compared to Spark UDAFs. Moreover, as expected, SUDAF shares the computation results of partial aggregates in every query sequence. For the sequence AS1, we observe in figure 2 (e) the computation times of count, variance (var), sum and average (avg) decrease drastically. This is because SUDAF is able to reuse cached results from earlier aggregates in the sequence AS1. The sequence AS2, shown figure 2 (f), is more advantageous for sharing due to the prefetched moment sketch. Indeed, the moment sketch consists of 33 partial aggregates which are cached and reused for the computation of all the remaining aggregations in the sequence AS2 except the harmonic mean (hm), which still requires data access since the aggregation state $\sum x_i^{-1}$ in hm is not evaluated in previous computations.

IV. Demonstration

We will present an end-to-end implementation of SUDAF and demonstrate its benefits. A video of our demonstration is available online [3]. In our demonstration, users can experience the *(i)* creation of UDAFs, *(ii)* executing queries with UDAFs and visualize their computation results and running times on SUDAF web user interface shown in figure 2. SUDAF web user interface is built on top of Apache Zeppelin. We will have a 6-node Spark cluster on our local platform (LIMOS Galactica [4]), and we will use TPC-DS dataset (scale = 100) and Milan Telecommunication dataset (319 million rows).

The two scenarios of SUDAF demo are detailed below.

(i) Creating UDAFs. Users can create and register UDAFs in SUDAF through two ways, either using SUDAF GUI or defining them programmatically. SUDAF GUI provides primitive functions supported in SUDAF shown in figure 2 (a), and users can compose primitive functions to create complex mathematical expressions. Alternatively, users can write a one-line Scala code to define UDAFs, e.g., defining geometric mean is shown in figure 2 (defining the same function in Spark SQL [4] is shown in figure 2(c)). Then, a UDAF can be used in SQL as shown in figure 2 (d).

(ii) Executing query sequences and visualizing running times. We provide a java object called workload, where users can continuously insert queries and run them using Spark SQL and SUDAF for an end-to-end comparison. We allow users to run queries having identical query model but different aggregations. One can run queries in arbitrary order and repeat them in arbitrary times. SUDAF caches partial aggregates and identifies sharing opportunities without any interference from users. The query results can be directly shown after their executions. Moreover, users can visualize execution times of individual queries in Spark SQL and SUDAF, e.g., the

[3] A demonstration video of SUDAF, https://www.youtube.com/watch?v=nH1S6PU6AK0. See also [8].

[4] https://galactica.isima.fr/

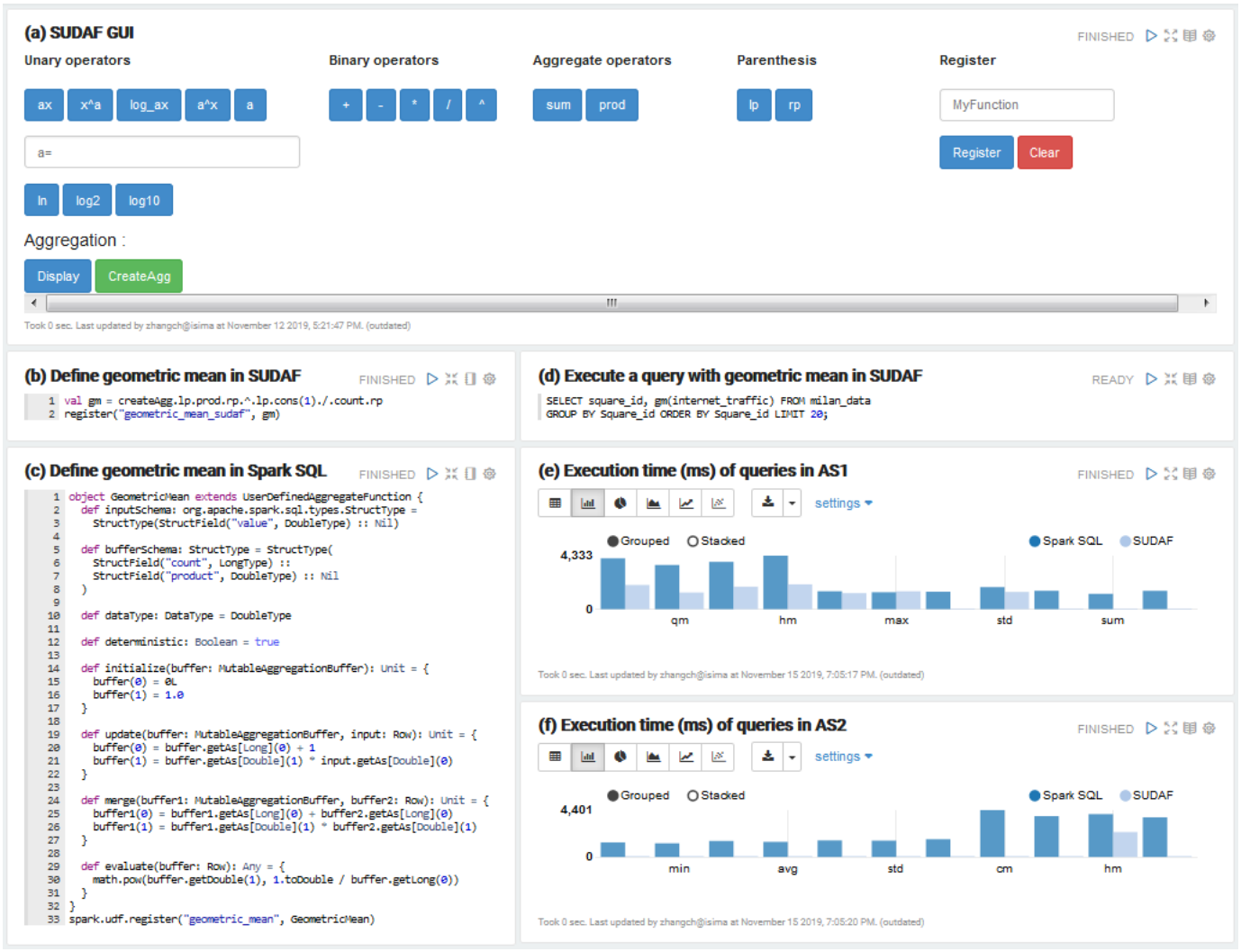

Fig. 2. Web user interface for SUDAF.

execution times of query sequences used in section III is shown in figure 2 (e) and (f). In this scenario, we will illustrate the two optimization approaches applied in SUDAF, rewriting partial aggregates of UDAFs using built-in functions and sharing partial aggregates of UDAFs. We will explain which approach is used in query processing by showing the query execution plan through Spark web UI.

V. Conclusion

We have introduced SUDAF, a framework that provides a set of primitive functions to enable users to declare their UDAFs. Our demonstration allows users to define mathematical expressions of UDAFs, run workloads of queries with UDAFs and visualize computation results and execution times of queries. The goal of this work is to provide a dynamic and efficient solution to accelerate queries with UDAFs. As future work, we envision to leverage the semantics of UDAFs captured by SUDAF to investigate query optimization and query rewriting using aggregate views.

Acknowledgment

This research was supported by the French government IDEXISITE initiative 16-IDEX-0001 (CAP 20-25) and by the ProFan project in the context of the “Programme dinvestissements davenir”.

References

[1] Apache Spark, https://spark.apache.org/.
[2] G. Edward, D. Jialin, T. Kai Sheng, S. Vatsal and B. Peter, “Moment-based quantile sketches for efficient high cardinality aggregation queries,” in VLDB 2018.
[3] JSqlParser, https://github.com/JSQLParser/JSqlParser.
[4] User defined aggregate functions - Scala, https://docs.databricks.com/spark/latest/spark-sql/udaf-scala.html.
[5] Telecommunications - SMS, Call, Internet - MI, https://doi.org/10.7910/DVN/EGZHFV.
[6] The technical report of SUDAF, https://github.com/CHAOHIT/SUDAF/blob/master/sudaf-technical-report.pdf.
[7] S. Cohen, “User-defined aggregate functions: bridging theory and practice,” in SIGMOD 2006.
[8] SUDAF demo video, https://www.youtube.com/watch?v=nH1S6PU6AK0, or https://github.com/CHAOHIT/SUDAF/blob/master/SUDAF-DEMO.mp4.
[9] PostgreSQL, https://www.postgresql.org/.