Sharing Computations for User-Defined Aggregate Functions

Chao Zhang
LIMOS, CNRS, University Clermont Auvergne
zhangchaohit13sg@gmail.com

Farouk Toumani
LIMOS, CNRS, University Clermont Auvergne
ftoumani@isima.fr

ABSTRACT

UDAFs (user-defined aggregate functions) are becoming a type of fundamental operators in advanced data analytics. The UDAF mechanism provided by most of the modern systems suffers, however, from at least two severe drawbacks: defining a UDAF requires hardcoding the routine that computes an aggregation, and the semantics of a UDAF is totally or partially unknown to the query processor, which hampers the optimization possibilities. This paper presents SUDAF (Sharing User-Defined Aggregate Functions), a declarative framework that allows users to write UDAFs as mathematical expressions and use them in SQL statements. SUDAF rewrites partial aggregates of UDAFs in users' queries using built-in aggregate functions and supports efficient dynamic caching and reusing of partial aggregates. Our experiments show that rewriting UDAFs using built-in functions can significantly speed up queries with UDAFs, and the proposed sharing approach can yield up to two orders of magnitude improvement in query execution time.

1 INTRODUCTION

An aggregate function has the inherent property of taking several values as input and generating a single value based on specific criteria [17, 25]. This ability to summarize information, the intrinsic feature of aggregation, has always been a fundamental task in data analysis [18, 24]. While earlier data management and analysis systems come equipped with a set of built-in aggregate functions, e.g., max, min, sum and count, it becomes clear that a limited set of predefined functions is not sufficient to cover the needs of the new applications in the age of analytics. In addition to augmenting the set of their built-in functions, most modern systems (e.g., [1, 2, 4, 21, 28, 29]) enable users to extend the system functionalities by defining their own aggregations. 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, i.e., 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 users to implement the routine that computes their aggregation functions. For example, to write a custom UDAF in Spark SQL [4], a user needs to map the UDAF to four methods: initialize, update, merge and evaluate, a.k.a. the IUME pattern. The user must ensure that the merge method is commutative and associative such that the UDAF can be computed correctly in a distributed architecture. In other words, to take benefit from distributed computations in Spark SQL, it is up to the user to identify whether her function supports partial aggregates (i.e., whether it is an algebraic function [18]). Secondly, the semantics of a UDAF, i.e., computation details, are not fully captured by a query engine, which hampers optimization possibilities. For example, when computing a UDAF that is created using the IUME pattern, a query engine can only be aware of calling an update function if there is a tuple or calling a merge function if there are intermediate results. However, the specific computations that are required to compute update and merge functions are unknown to a query engine since these two functions are hardcoded. The loss of such computation details prevents a query engine from sharing partial results of different UDAFs.

In the context of aggregate queries optimization, materialized views with aggregates or cached queries are among the techniques that can be used to accelerate query processing. In this context, most existing works focus on the data dimension [8, 11, 12, 15], i.e., sharing identical aggregates computed over overlapping range predicates or different data granularities. Admittedly, considering only the data dimension restricts the sharing possibilities to queries with identical aggregation operators. To cope with such a limitation, few works propose to use predefined rules to specify how a given aggregate can be computed from the results of another one [10, 33]. However, such a static approach requires one to explicitly predefine the computation rules across prefixed aggregates, which hinders the optimization for UDAFs defined on the fly.

The objective of this work is twofold: firstly, we aim at giving full flexibility to users by providing a declarative framework that allows them to write UDAFs as mathematical expressions and use them in SQL queries[1]. Then, a UDAF is decomposed into partial aggregates, which are then rewritten using built-in functions, i.e., scalar functions and aggregations. Secondly, our goal is to develop a *dynamic* approach for caching and reusing partial aggregates of UDAFs to optimize the computations of UDAFs. More precisely, we aim at identifying when it is possible to reuse cached partial aggregates of past UDAFs to compute new UDAFs.

Contributions. Our main contributions, implemented in the SUDAF framework, are as follows:

- We present SUDAF, a declarative UDAF framework that allows users to formulate a UDAF as a mathematical expression and use them within SQL queries. When executing a given query with UDAFs, SUDAF identifies appropriate partial aggregations from the mathematical expression of a UDAF and rewrites them using built-in functions of an underlying data management and analysis system.
- We formalize the problem of identifying when a partial aggregate of a given UDAF can be used in the computation of another UDAF as the *sharing problem*, and we show that this problem is undecidable in a general setting.
- To deal with the undecidability of the *sharing problem*, we restrict the set of UDAFs supported in SUDAF by providing classes of primitive functions that can be used to describe mathematical expressions of UDAFs. This practical framework is powerful enough to be used in practical applications. From a theoretical standpoint, we characterize

[1]This approach is more intuitive than programming the procedure of an aggregation, e.g., Wolfram Mathematica provides mathematical expressions to define advanced statistical computation [34].

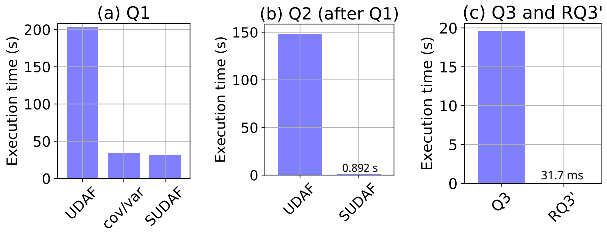

Figure 1: Experiments in PostgreSQL with the TPC-DS dataset (scale = 20). UDAFs theta1() and qm() are created in PL/pgSQL.

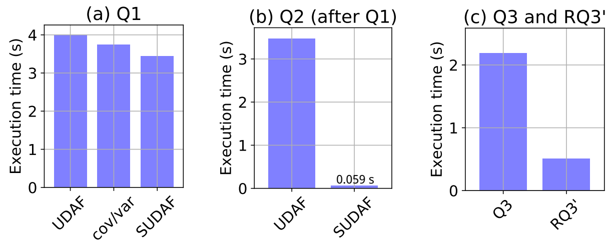

Figure 2: Experiments in Spark SQL with the TPC-DS dataset (scale = 100). UDAFs theta1() and qm() are created using UserDefinedAggregateFunction in Scala.

the sharing problem in SUDAF and provide corresponding sharing conditions (Theorem 4.1). From a practical standpoint, we design an approach based on symbolic representations of mathematical expressions to efficiently verify the proposed conditions.

- We implemented a SUDAF prototype and report on experiments using SUDAF with both PostgreSQL and Spark SQL. Our experiments show that rewriting partial aggregates of UDAFs using built-in aggregates can significantly speed up query execution. In addition, the proposed sharing technique can yield up to two orders of magnitude improvement in query execution time.

The rest of this paper is organized as follows. We present a motivating example to illustrate SUDAF's main features in Section 2. In Section 3, we introduce a canonical form of UDAFs and discuss the sharing problem in this context. In Section 4, we present the SUDAF framework and show that the sharing problem is decidable in this context. In Section 5, we introduce a practical approach, based on symbolic representations of partial aggregates, to solve the sharing problem in the SUDAF framework. In Section 6, we present an experimental evaluation of SUDAF. We discuss related works in Section 7 and conclude in Section 8. All related proofs are included in our online technical report [31].

2 MOTIVATING EXAMPLE

In this section, we present a motivating example demonstrating two SUDAF's functionalities: (i) rewriting UDAFs using built-in functions, and (ii) sharing partial aggregation results between different UDAFs. In addition, we also illustrate how the sharing mechanism can be used to extend query rewriting using aggregate views. In the following example, we consider 4 relations of the TPC-DS [27] dataset, store_sales, store, date_dim and stores.

Suppose that a user wants to analyze the price of every item sold by the stores in the state Tennessee (TN) in the past every year. Specifically, the user has a hypothesis of a *simple linear regression*: $y = \theta_1 x + \theta_0$, where y represents a value in the sales_price column and x a value in the list_price column. Using the least square error function, we have $\theta_1(X,Y) = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$, and $\theta_0(X,Y) = avg(Y) - \theta_1 avg(X)$.

One can hardcode θ_1 as a user-defined function and then uses it in an SQL statement, e.g., one writes a piece of Java or Scala code to create θ_1 in Spark SQL (see Scala code in [31]). Assume that a hardcoded user-defined function theta1(), that implements the function $\theta_1()$, is created and the following query Q1 is issued:

```
Q1: SELECT  ss_item_sk, d_year, avg(ss_list_price),
            avg(ss_sales_price),
            theta1(ss_list_price,ss_sales_price)
    FROM    store_sales, store, date_dim
    WHERE   ss_sold_date_sk = d_date_sk and
            ss_store_sk = s_store_sk and s_state = 'TN'
   GROUP BY ss_item_sk, d_year;
```

Alternatively, in SUDAF the function theta1() is defined declaratively by providing its mathematical expression without the needs of any programming effort.

Now, assume that a user defines the expressions of theta1() and avg() and uses them in the query Q1. We illustrate in the rest of this section two benefits of using SUDAF to execute the query Q1: (i) the partial aggregates of theta1() and avg() used in the query Q1 are rewritten into a set of partial aggregates using the built-in functions *sum* and *count*, and (ii) the partial aggregates computed during the execution of Q1 can be cached and reused to compute various other UDAFs.

Rewriting partial aggregates using built-in functions. The first step of processing Q1 in SUDAF is to factor out partial aggregates of theta1() and avg() and rewrite them using built-in functions to compute. 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, $\theta_1 = \frac{s_1 s_5 - s_4 s_2}{s_1 s_3 - (s_2)^2}$.

```
RQ1: SELECT ss_item_sk, d_year, s2/s1 avg_list_price,
            s4/s1 avg_sales_price,
            (s1*s5-s4*s2)/NULLIF((s1*s3-power(s2,2)),0) theta1
     FROM   (SELECT ss_item_sk, d_year, count(*) s1,
                    sum(ss_list_price) s2,
                    sum(power(ss_list_price,2)) s3,
                    sum(ss_sales_price) s4,
                    sum(ss_sales_price*ss_list_price) s5
             FROM   store_sales, store, date_dim
             WHERE  ss_sold_date_sk = d_date_sk and
                    ss_store_sk = s_store_sk and
                    s_state = 'TN'
             GROUP BY ss_item_sk, d_year) TEMP;
```

Compared to the original query Q1, RQ1 uses only built-in aggregate functions and hence it is expected to be much more efficient because built-in functions are better handled by existing query optimizers and execution engines than hardcoded user-defined functions. Figure 1 (a) shows that the execution of Q1 using SUDAF on top of PostgreSQL can be $10X$ faster compared to running Q1 directly over PostgreSQL. Similar results can be observed in Figure 2 (a) using SUDAF on top of Spark SQL, where Q1 is $1.25X$ faster compared to the direct execution of Q1 over Spark SQL. To be fair in our analysis, we should mention that in the context of PostgreSQL and Spark SQL systems, where the covariance (*cov*) and the variance (*var*) are built-in functions, an alternative and efficient implementation of theta1() can be obtained using the formula theta1() = cov/var. We also report the query time of using *cov*/*var* in Q1, respectively in Figure 1 (a) and Figure 2 (a), which is at the same order of magnitude as SUDAF execution time. However, even in this case, the benefit of using SUDAF comes from the fact that the performance of SUDAF is independent of the user's programming skill and, as shown in the next example, the partial aggregates computed by SUDAF using sum and count aggregates open wider sharing possibilities than the variance and covariance functions.

Note that SUDAF decomposes a UDAF into two parts, a set of partial aggregates and a terminating function T, then only the partial aggregates of a UDAF are rewritten using built-in functions. This is because a terminating function T is essentially a scalar function applied only on several partial aggregates, and hence it does not impact the computation time of a UDAF. Moreover, there are some UDAFs where it is not possible to write their corresponding terminating functions using built-in functions, e.g., the MomentSolver [16] used to approximate a quantile.

Sharing partial aggregates across UDAFs. Caching the result of Q1, which contains the aggregate values of theta1(), is of little interest from the sharing perspective. However, the partial aggregates $s_1, \ldots, s_5$ computed by the query RQ1 offer more possibilities to be reused in future UDAF computations. We illustrate the sharing idea by the following example. Consider a new query Q2 that computes quadratic mean qm() and standard deviation stddev() of list prices of every item sold by stores in TN for every year:

```
Q2: SELECT  ss_item_sk, d_year, qm(ss_list_price),
            stddev(ss_list_price)
    FROM    store_sales, store, date_dim
    WHERE   ss_sold_date_sk = d_date_sk and
            ss_store_sk = s_store_sk and s_state = 'TN'
    GROUP BY ss_item_sk, d_year;
```

Using SUDAF, qm() (an instance of power mean with $p = 2$ shown in Table 1) and stddev() are defined using the mathematical expressions given in Table 1. When executing Q2, SUDAF factors out their partial aggregations and generates the following query RQ2 which uses the same partial aggregates s_1, s_2 and s_3 as the query RQ1.

```
RQ2: SELECT ss_item_sk, d_year, sqrt(s3/s1) qm_list_price,
            sqrt(s3/s1-power(s2/s1,2)) std_list_price
     FROM   (SELECT ss_item_sk, d_year, count(*) s1,
                    sum(ss_list_price) s2,
                    sum(power(ss_list_price,2)) s3
             FROM   store_sales, store, date_dim
             WHERE  ss_sold_date_sk = d_date_sk and
                    ss_store_sk = s_store_sk and
                    s_state = 'TN'
             GROUP BY ss_item_sk, d_year) TEMP2;
```

SUDAF can cache the partial aggregates in the query RQ1 and identify the opportunity to reuse them for computing aggregates in the query RQ2 automatically. This makes the execution of Q2 in SUDAF significantly faster than executing the query Q2 from base data. We report the query time of Q2 when it is executed by SUDAF on top of PostgreSQL in Figure 1 (b) and on top of Spark SQL in Figure 2 (b). In both figures, the execution time of SUDAF is compared to the execution time of the query Q2 computed respectively over PostgreSQL and Spark SQL. We would like to stress the fact that the result of the UDAF theta1() computed by the query RQ1 cannot be reused to compute the UDAF qm() and stddev() of the query RQ2. However, identifying the appropriate partial aggregates of RQ1 and RQ2 enables to increase the sharing opportunities between these two queries.

Note that we only consider in our example the computation dimension, i.e., computing a UDAF from other UDAFs. Full implementation of our approach requires handling the data dimension, i.e., whether a query is semantically contained in the cached query, which is not addressed in this paper. We point out existing techniques [15, 33] based on data partitioning that can be used in our context to handle the data dimension issue. The main idea of such techniques is to partition the data into predefined chunks and then to map a given query to chunks. Extending SUDAF with such techniques enables us to share partial aggregates over predefined data chunks.

We would like to stress the following three features of the SUDAF sharing mechanism:

- Firstly, it increases performance significantly compared to SUDAF without sharing. In this example, using SUDAF without sharing over PostgreSQL to compute Q2 will take 33.61 s, which is far slower compared to 0.892 s shown in Figure 1 (b). Similarly, in the case of using SUDAF over SparkSQL, SUDAF without sharing will take 2.953 s, which is also significantly slower compared to 0.059 s shown in Figure 2 (b).
- Moreover, the sharing opportunity is dynamically identified in SUDAF by analyzing the expressions of partial aggregates in UDAFs. Note that, using a static approach, one has to predefine computation rules for specific aggregations, e.g., defining $stddev \rightarrow s_1, s_2, s_3$ to share results between RQ1 and RQ2, which is not required in SUDAF.
- Finally, the sharing mechanism of SUDAF covers also the case where partial aggregates are not identical (we present sharing conditions in Section 4.2). For example, SUDAF enables sharing computations between geometric mean and the aggregate $\sum ln(x_i)$, an element of the moment sketch [16]. This is because the partial aggregate $\prod x_i$ of geometric mean (see Table 1) can be computed from $\sum ln(x_i)$, i.e., $\prod x_i = exp(\sum ln(x_i)), \forall x_i > 0$ (see detailed experiments in Section 6).

Extending query rewriting using aggregate views. We show that factoring out partial aggregations of UDAFs can improve traditional query rewriting using aggregate views. Assuming a user is interested in computing qm() and stddev() of the list prices of all items in the category of sports sold by stores in TN for every year since 2000. This is expressed by the following query Q3.

```
Q3: SELECT  d_year, qm(ss_list_price), stddev(ss_list_price)
    FROM    store_sales, store, date_dim, item
    WHERE   ss_sold_date_sk = d_date_sk and ss_item_sk =
            i_item_sk and ss_store_sk = s_store_sk and
            i_category = 'Sports' and s_state = 'TN'
            and d_year >= 2000
    GROUP BY d_year;
```

Now, assume that a materialized view VQ1 corresponding to the query Q1 is given. One can realize that the view VQ1 is useless for rewriting Q3 since it is not possible to compute qm() and stddev() from theta1() and avg().

However, if a materialized view V1 corresponding to the subquery of RQ1 is given and if we factor out partial aggregations of qm() and stddev() in Q3 to generate the following query RQ3:

```
RQ3: SELECT d_year, sqrt(s3/s1) qm_list_price,
            sqrt(s3/s1-pow(s2/s1,2)) std_list_price
     FROM   (SELECT  d_year, count(*) s1,
                     sum(ss_list_price) s2,
                     sum(power(ss_list_price,2)) s3
             FROM    store_sales, store, date_dim, item
             WHERE   ss_sold_date_sk = d_date_sk and
                     ss_item_sk = i_item_sk and
                     ss_store_sk = s_store_sk and
                     i_category = 'Sports'
                     and s_state = 'TN'and d_year >= 2000
             GROUP BY d_year) TEMP3;
```

Then it is possible to use the rewriting algorithm proposed in [13] to rewrite the subquery of RQ3 using V1. The obtained rewriting, denoted by RQ3', is shown below.

Table 1: Examples of aggregations in canonical forms.

Aggregation	Expression	Canonical form $(\mathrm{F}, \oplus, \mathrm{T})$
Power mean	$(\frac{\sum(x_i)^p}{n})^{1/p}$	$\left((1, x_i^p), (+,+), (\frac{s_2}{s_1})^{1/p}\right)$
Geometric mean	$(\prod x_i)^{1/n}$	$\left((x_i, 1), (\times,+), (s_1)^{1/s_2}\right)$
Stddev	$\sqrt{\frac{\sum x_i^2}{n} - (\frac{\sum x_i}{n})^2}$	$\left((1, x_i, x_i^2), (+,+,+), \sqrt{\frac{s_3}{s_1} - (\frac{s_2}{s_1})^2}\right)$
Central moment	$\frac{\sum(x_i - avg)^k}{n}$	$\left(((x_i - avg)^k, 1), (+,+), s_1/s_2\right)$
LogSumExp	$ln(\sum exp(x_i))$	$\left((exp(x_i)), (+), ln(s_1)\right)$

Aggregation	Expression	Canonical form $(\mathrm{F}, \oplus, \mathrm{T})$
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	$\frac{\sum(x_i y_i)}{n} - \frac{\sum x_i \sum y_i}{n^2}$	$\left((x_i, y_i, x_i y_i, 1), (+,+,+,+), \frac{s_3}{s_4} - \frac{s_1 s_2}{s_4}\right)$
Correlation	$\frac{n\sum(x_i y_i) - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$	$\left((x_i, x_i^2, y_i, y_i^2, x_i \times y_i, 1), (+,+,+,+,+,+), \frac{s_6 s_5 - s_1 s_3}{\sqrt{s_6 s_2 - (s_1)^2}\sqrt{s_6 s_4 - (s_4)^2}}\right)$

```
RQ3': SELECT d_year, sqrt(s3/s1) qm_list_price,
             sqrt(s3/s1-pow(s2/s1,2)) std_list_price
      FROM   (SELECT  d_year, sum(s1) s1, sum(s2) s2,
                      sum(s3) s3
              FROM    V1, item
              WHERE   ss_item_sk = i_item_sk and
                      d_year >= 2000 and
                      i_category = 'Sports'
              GROUP BY d_year) TEMP3;
```

The key reason that enables such a rewriting comes from the fact that the UDAFs have been rewritten using built-in aggregates: sum() and count() (we recall that the rewriting algorithm proposed in [13] supports only the *sum* and *count* aggregates). We report the execution time of Q3 and RQ3' in PostgreSQL in Figure 1 (c) and Spark SQL in Figure 2 (c).

To conclude this section, we would like to emphasis the fact that the main features of SUDAF, factoring out the partial aggregations of UDAFs, computing partial aggregations using built-in functions and sharing partial aggregates, provide abundant opportunities to speed up queries with UDAFs. In the rest of this paper, we address the following challenges:

- *how to identify appropriate partial aggregations of UDAFs to maximize sharing opportunities?*
- *how to efficiently determine when cached results of partial aggregations of UDAFs can be reused to compute other UDAFs*? (hereafter, called the sharing problem)

3 IDENTIFYING AND SHARING PARTIAL AGGREGATES

We aim at speeding up queries with UDAFs by reusing cached answers to previous queries with UDAFs during the evaluation of new ones. We deal with the following two issues in this section.

What computation results should be cached to optimize the evaluation of UDAFs? We identify a canonical form of UDAFs [10], which captures the computation pipelines of UDAFs. We analyze the caching possibilities based on the computation pipelines and identify the appropriate level of aggregation to be kept in caches.

How can we identify if a cached answer can be reused in the evaluation of a given UDAF? We formalize the problem of identifying a reusable answer as the sharing problem. Then we show that it is an undecidable problem for arbitrary cases. In Section 4, we present a restricted, yet powerful enough, framework to handle the sharing problem for practical cases.

3.1 Canonical forms of UDAFs

An aggregate function takes as inputs several values and produces as output a *single representative* value [17]. In our work, we consider aggregations operating on multisets. Let D_s and D_t be two domains i.e., countably infinite sets of values, and let $\mathcal{M}(D_s)$ denote the set of all nonempty multisets of elements from D_s. An aggregate function α is a function: $\mathcal{M}(D_s) \rightarrow D_t$.

We use the notion of well-formed aggregation to define a canonical form of aggregate functions. Well-formed aggregation was introduced in [10] to capture the manner in which a UDAF is created. An aggregation $\alpha : \mathcal{M}(D_s) \rightarrow D_t$ is a *well-formed aggregation* if α can be expressed as a triple $(F, \oplus, T)$, where F is a translating function, $\oplus$ is a commutative and associative binary operation and T is a terminating function, such that $\forall X = \{\!\{x_1, ..., x_n\}\!\} \in \mathcal{M}(D_s), \alpha(X) = T(F(x_1) \oplus \ldots \oplus F(x_n))$, or briefly $\alpha(X) = T(\sum_{\oplus} F(x_i))$.

In this paper, we consider the well-formed aggregation as the canonical form of UDAFs. We list some examples of aggregations with their canonical forms in Table 1 (an input of a terminating function T is denoted as s_i). It is interesting to note that practical aggregations usually have addition and multiplication as an element of $\oplus$ function in their canonical forms, e.g., the $\oplus$ function of geometric mean is $(\times, +)$.

Given an aggregation, $\alpha = (F, \oplus, T)$, the associative and commutative property of $\oplus$ ensures that $\alpha(X)$ can be computed by first applying F and $\oplus$ on arbitrary subsets of X and then the intermediate results can be merged using $\oplus$ and T to produce the final result $\alpha(X)$. Hence, we call the intermediate results $\sum_{\oplus} F(x_i)$ the *partial aggregations* of α.

3.2 Caching aggregate data

To obtain more sharing possibilities, we identify which results of an aggregation are worth caching based on its canonical form. Consider two aggregations $\alpha = (F_\alpha, \oplus_\alpha, T_\alpha)$ and $\beta = (F_\beta, \oplus_\beta, T_\beta)$. Suppose a scenario where an implementation of α based on its canonical form is executed first. When the UDAF β is evaluated, there are three possibilities to reuse partial or whole computation results of α: (1) the result of F_α, (2) the result of $\sum_{\oplus_\alpha} F_\alpha$, or (3) the final result of α. It is clear that caching the 1st result does not provide any added value to the computation of β since F_α is a scalar function. Storing the 3rd result is of little interest as it offers very restricted possibilities [2] to be reused in the computation of other UDAFs, e.g., β. However, the partial aggregation $\sum_{\oplus_\alpha} F_\alpha$ offers much more potentials to reuse than the others. For example, if α is a *stddev* and β is a *power mean* ($p = 2$) shown in Table 1, it is not possible to reuse the final result of α to compute β. However, using their canonical forms, one can observe that the fragments, s_1 and s_3, in the partial aggregation of α can be used to compute β. Therefore, we choose to cache the partial aggregation $\sum_{\oplus_\alpha} F_\alpha(x_i)$.

[2]Theoretically, T_α should not be expected to have an inverse function [10], such that we cannot always have the 2nd result if we cache the 3rd one. However, we can indeed have the 3rd result if we cache the 2nd result.

Table 2: Classes of primitive functions provided in SUDAF.

Class	Functions
PS	a; x; ax; x^a; $log_a x$; a^x.
PB	$+$; $-$; $\times$; $/$; ^.
PA	$\sum$; $\prod$.
PS°	$g(x) = h_l \circ \ldots \circ h_1(x)$, with $h_j \in PS$, for $j \in [1, \ldots, l]$.
$PS^{\odot}$	$f(x) = g_k(x) \odot_{k-1} \ldots \odot_1 g_1(x)$, with $g_j \in PS^{\circ}, \odot_z \in PB$, for $j \in (1, \ldots, k), z \in (1, \ldots, k-1), k \in \mathbb{N}_{>0}$.
PA°	$agg(X) = f' \circ \sum_{\oplus} \circ f(x_i)$, with $f, f' \in PS^{\odot}, \sum_{\oplus} \in PA$.
$PA^{\odot}$	$bagg(X) = T'(agg_k(X) \odot_{k-1} \ldots \odot_1 agg_1(X))$, with $agg_j \in PA^{\circ}$,$\odot_z \in PB$ for $j \in (1, \ldots, k)$,$z \in (1, \ldots, k-1), k \in \mathbb{N}_{>1}$ and $T' \in PS^{\odot}$.

Table 3: Cases analysis of the sharing problem in SUDAF.

Case	f_1 in s_1	f_2 in s_2	Whether $s_1 \in D(s_2)$
1	Injective	Non-injective	N (case 1 of Theorem 4.1)
2	-	Injective	Case 2 of Theorem 4.1
3	Even	Even	Case 2 of Theorem 4.1
4	Neither injective nor even	Neither injective nor even	Splitting rules (SR)

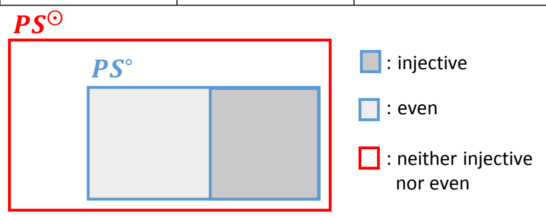

Figure 3: Injective and even functions in PS° and $PS^{\odot}$ (excluding constant functions).

3.3 Sharing aggregation states

Let $\alpha = (F, \oplus, T)$ be an aggregation and $\sum_{\oplus} F(x_i)$ be the partial aggregation of α. We decompose the partial aggregation as follows, $\sum_{\oplus} F(x_i) = \left(\sum_{\oplus_1} f_1(x_i), \ldots, \sum_{\oplus_m} f_m(x_i) \right)$, where the f_is are scalar functions and the $\oplus_i$s are commutative and associative binary operations, e.g., the partial aggregation of geometric mean is $(\prod x_i, count)$. In the sequel, we call an individual element $s_j(X) = \sum_{\oplus_j} f_j(x_i)$ as an *aggregation state*, e.g., both $\prod x_i$ and *count* are aggregation states of geometric mean.

We rely on aggregation states to define when a partial result of a UDAF α can be reused in the computation of another UDAF β. More precisely, we define below when an aggregation state s of α can be *shared* by an aggregation state s' of β.

Definition 3.1. Let $s'(X)$ and $s(X)$ be two aggregation states of two UDAFs. Then, s' shares s iff there exists a computable function r such that $s'(X) = r \circ s(X), \forall X \in \mathcal{M}(D)$.

The function r is a scalar function that enables computing the aggregation state s' without scanning the base dataset X, e.g., r is the identity function if $s'(X) = s(X)$. If an aggregation state s is cached, the sharing problem is then to decide whether s can be reused in the computation of another aggregation state s'.

We denote the problem whether s' shares s as share(s', s). As stated by the following theorem, it is not possible to solve share(s', s) in a general setting. The proof for Theorem 3.2 is included in our online technical report [31].

Theorem 3.2. *The problem share(s', s) is undecidable.*

4 THE SUDAF PRACTICAL FRAMEWORK

In this section, we present a declarative UDAF framework SUDAF, which rests on the canonical form of UDAFs to generate and share partial aggregation states of UDAFs automatically. The following main objective guided the design of SUDAF.

How to deal with the undecidability of the sharing problem? We adopt a pragmatic approach to solve this problem by restricting the class of UDAFs that can be used in SUDAF. The proposed practical framework is powerful enough to be useful in many real-world applications while making the sharing problem decidable.

We argue that it is not realistic to ask a user to provide UDAFs in their canonical forms. Therefore, SUDAF enables users to formulate UDAFs as mathematical expressions and then generates a corresponding canonical form. Consequently, in a generated canonical form, SUDAF knows the semantics of partial aggregations, i.e., computation details, which can be exploited to analyze sharing possibilities during computing UDAFs.

4.1 Declarative UDAF framework

SUDAF provides a set of predefined functions that can be used by users to write UDAFs. Three classes of primitive functions are proposed (cf. Table 2):

- *Primitive scalar functions.* This class, denoted PS (primitive scalar), contains six types of functions: constant, identity, linear, power, logarithmic and exponential functions. The elements of PS are presented in line 1 of Table 2, where a is an arbitrary constant defined by users.
- *Primitive binary functions.* This class, denoted PB (primitive binary), contains the following binary functions: addition $+$, subtraction $-$, multiplication $\times$, division $/$ and exponentiation ^.
- *Primitive aggregate functions.* This class, denoted PA (primitive aggregate functions), contains two functions: summation $\sum$ and product $\prod$.

As explained below, primitive functions can be combined using the composition operator and binary functions to create more complex scalar and aggregate functions.

Complex scalar functions. SUDAF provides a *composition operator*, denoted $\circ$, that enables creating complex scalar functions from the primitive ones. The class of such functions is denoted PS°. A function $g(x) \in PS^{\circ}$ can be expressed as a composition of primitive scalar functions (cf. Table 2). The length of $g(x)$, denoted $|g|$, gives the number of primitive functions used in the definition of $g(x)$. For example, if $g(x) = h_l \circ \ldots \circ h_1(x)$, with $h_j \in PS$, then $|g| = l$. Besides, more complex scalar functions can be expressed by using binary functions to combine scalar functions from PS°. The set of such functions, i.e., scalar functions containing binary operations, is denoted $PS^{\odot}$. The shape of functions in $PS^{\odot}$ is shown in Table 2.

Supported aggregations. SUDAF also allows using the composition operator $\circ$ between scalar functions and primitive aggregate functions to define new aggregations. More precisely, in this context, the composition can be used in two ways: *(i)* to apply a scalar function on an output of a primitive aggregate function, or *(ii)* to apply a primitive aggregation on a set of data transformed using a scalar function. The class of such functions is denoted as PA°. The expression of aggregation $agg \in PA^{\circ}$ is presented in Table 2. Moreover, more complex aggregations can be expressed using primitive binary functions to combine several aggregations in PA°. The class of such functions is denoted as $PA^{\odot}$, and a UDAF $bagg \in PA^{\odot}$ has the expression shown in Table 2.

Scope of UDAFs in SUDAF. SUDAF restricts the set of UDAFs that can be declared to the classes presented in Table 2. We shall show in the next section that this restriction enables us to cope with the undecidability of sharing problems. However, this restriction does not hamper the usability of SUDAF in real world applications since the proposed framework covers a wide range of aggregations such as the classes of power mean, arbitrary central moments [7], arbitrary standardized moments [32] and other multi-variate aggregations [3] such as covariance, correlation, and cofactor aggregates [30] used in training linear regression. Generally, algebraic aggregations can be defined in SUDAF. Although holistic aggregations, e.g., median, cannot be expressed in SUDAF, aggregates used in their approximation algorithms are supported by SUDAF, e.g., moment sketch [16],

Mapping SUDAF functions into canonical forms. SUDAF supports two scenarios to define UDAFs. We explain below how to derive canonical forms and aggregation states from UDAFs defined in each scenario.

The first scenario is that a terminating function is described using an element from $PS^{\odot}$. Such functions are expressed using a function $T' \in PS^{\odot}$ applied on compositions, using binary operations in PB, of aggregations from PA° and have the following general form:

$$\alpha(X) = T'\big(\big(f'_k \circ \sum_{\oplus_k} \circ f_k(x_i)\big) \odot_{k-1} \dots \odot_1 \big(f'_1 \circ \sum_{\oplus_1} \circ f_1(x_i)\big)\big).$$

The f_j, f'_j, for $j \in [1, ..., k]$, are scalar functions from $PS^{\odot}$ and $\sum_{\oplus_j}$ are primitive aggregations from PA. Given such a function $\alpha(X) \in PA^{\odot}$, a canonical form canonical$(\alpha) = (F, \oplus, T)$ is derived from the general expression of α as follows:

- $F = (f_1, \dots, f_k)$;
- $\oplus = (\oplus_1, \dots, \oplus_k)$ and
- $T = T'\big((f'_1 \circ \sum_{\oplus_1} \circ f_1) \odot_1 \dots \odot_{k-1} (f'_k \circ \sum_{\oplus_k} \circ f_k)\big)$.

The aggregation states of α are shown as follows: $s_j(X) = \sum_{\oplus_j} f_j(x_i)$, for $j \in [1, \dots, k]$. For instance, aggregations in Table 1 can be defined in SUDAF using their expressions in the second column. SUDAF generates their canonical forms and aggregation states from their expressions (the s_i elements in Table 1).

The second scenario is that a terminating function is created by hardcoding. Such functions have the following shapes, $\alpha(X) = T(s_1, ..., s_k)$, where $s_j, j \in (1, ..., k)$ is an aggregation state. For example, if one wants to use the MomentSolver [16] taking the MomentSketch as inputs to approximate a quantile, the MomentSketch can be defined as a set of aggregation states from PS° and the MomentSolver as a terminating function.

4.2 Dealing with the sharing problem in SUDAF

In this section, we present sharing conditions to deal with the sharing problem in SUDAF. Let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states of two UDAFs in the scope of SUDAF. Then both f_1 and f_2 belong to $PS^{\odot}$. We carry out a case analysis to identify the conditions that characterize situations where s_1 shares s_2. Our case analysis is based on the properties of the scalar functions f_1 and f_2 used by the aggregation states s_1 and s_2. In fact, all scalar functions in PS°, except constant functions, are either injective, or even (i.e., $f(x) = f(-x)$)), while scalar functions in $(PS^{\odot} \setminus PS^{\circ})$ are not injective because of the presence of the arithmetic binary functions $\odot$ (cf. Figure 3). Therefore, we split the *sharing problem* share(s_1, s_2) into four main cases depending on whether f_1 and f_2 are injections or even functions. The studied cases are presented in Table 3. Our main results provide a full characterization for the first three cases in Table 3. Specifically, we provide complete conditions in Theorem 4.1 for the first two cases in Table 3, and then we reduce the third case to the second case in Table 3. We also propose an incomplete solution to deal with the fourth case in Table 3.

Theorem 4.1. *Let $X \in \mathcal{M}(\mathbb{Q})$ and let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states with $\sum_{\oplus_1} \in PA$ and $\sum_{\oplus_2} \in PA$, f_1 a non constant function and $s_1 \neq s_2$. Then, we have:*

(Case 1) *if f_1 is injective and f_2 is not injective, then s_1 does not share s_2.*

(Case 2) *if f_2 is injective, then: there exists a computable function r_{12} such that $s_1(X) = r_{12} \circ s_2(X)$ iff one of the following conditions holds:*

(2.1) *$\sum_{\oplus_1} = \sum_{\oplus_2} = \sum$ and $f_1 \circ f_2^{-1}(x) = ax$ with $a \in \mathbb{Q}_{\neq 0}$ a constant. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.2) *$\sum_{\oplus_1} = \sum$, $\sum_{\oplus_2} = \prod$ and $f_1 \circ f_2^{-1}(x) = a(log_b|x|)$ with $b \in \mathbb{Q}_{>0,\neq 1}$ and $a \in \mathbb{Q}_{\neq 0}$ two constants. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.3) *$\sum_{\oplus_1} = \prod$, $\sum_{\oplus_2} = \sum$ and $f_1 \circ f_2^{-1}(x) = b^{ax}$ with $b \in \mathbb{Q}_{>0,\neq 1}$ and $a \in \mathbb{Q}_{\neq 0}$ two constants. Then we have $r_{12}(x) = f_1 \circ f_2^{-1}(x)$.*

(2.4) *$\sum_{\oplus_1} = \sum_{\oplus_2} = \prod$ and with a constant $a \in \mathbb{Q}_{\neq 0}$:*
(i) *when $f_1 \circ f_2^{-1}(-1) = 1$, $f_1 \circ f_2^{-1}(x) = |x|^a$;*
(ii) *when $f_1 \circ f_2^{-1}(1) = -1$, $f_1 \circ f_2^{-1}(x) = sgn(x) \times |x|^a$; Then we have $r(x) = f_1 \circ f_2^{-1}(x)$.*

The proof for Theorem 4.1 is included in a technical report [31].

The case 1 of Theorem 4.1 states that, given two aggregation states $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ in the scope of SUDAF, when f_1 is injective and f_2 is non-injective, then except the special case of an identity function when $s_1 = s_2$, it is not possible to find a computable function r_{12} such that $s_1(X) = r_{12} \circ s_2(X)$. The case 2 of Theorem 4.1 provides necessary and sufficient conditions to characterize solutions for the problem share(s_1, s_2) when f_2 is injective. It carries out a case analysis for the four possible combinations obtained from the instantiation of $\sum_{\oplus_1}$ and $\sum_{\oplus_2}$ as operations in PA, i.e., either sum or product.

Example 4.2. We explain how Theorem 4.1 can be used as follows. Consider the problem whether $s_1(X) = \sum 4x_i$ shares $s_2(X) = \prod 2^{x_i}$. Since $\sum_{\oplus_1} = \sum$ and $\sum_{\oplus_2} = \prod$, then the case 2.2 of Theorem 4.1 is selected. Then, we have $f_1 \circ f_2^{-1}(x) = 4log_2(x)$, which satisfies the shape $a(log_b(x))$ with constants $a = 4$ and $b = 2$. Thus, we have $s_1(X) = r \circ s_2(X)$ with $r(x) = 4log_2(x)$.

The case of even scalar functions. The third case to deal with is when both $f_1(x)$ and $f_2(x)$ are not injections but even functions (case 3 of Table 3). As depicted in Figure 3, non-injective scalar functions of PS° are *even* functions. We exploit this property to reduce the study to a sharing problem over a positive domain of scalar functions and show that the case 2 of Theorem 4.1 can be applied in this setting. We denote $U_X = \{u_x = |x| | x \in X\}$.

[3] Multi-variate aggregations can be seen as a combination of several uni-variate aggregations, each of which is expressed using functions in Table 2. Moreover, the cofactor aggregate $\sum x_i y_i$ computed over columns X and Y can be seen as a uni-variate aggregate over an abstract column $Z = X \cdot Y$ with the scalar product $\cdot$.

Then, whatever x is, we have $u_x \geqslant 0$. Let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states in SUDAF such that $\{f_1, f_2\} \subset PS^{\circ}$. Observe that $s_1(X)$ shares $s_2(X)$ iff $s_1(U_X)$ shares $s_2(U_X)$. This is because $f_1(x) = f_1(u_x)$ (since f_1 is even), and similarly for f_2. Consequently, one can focus on solving the sharing problem only over positive domains of f_1 and f_2. In this setting (positive domain), all primitive scalar functions of SUDAF (non-constant elements in PS) are injections and hence the complex scalar functions, elements of PS°, are also injective functions. Therefore, the case 2 of Theorem 4.1 can be exploited to solve the sharing problem in this context.

The case of neither even nor injective scalar functions. The last case to deal with is when both $f_1(x)$ and $f_2(x)$ are neither injections nor even functions (case 4 of Table 3). As depicted in Figure 3, such scalar functions are from $(PS^{\odot} \setminus PS^{\circ})$. We propose splitting rules to deal with such cases. W.l.o.g, let $s(X) = \sum_{\oplus}(g_1(x_i) \odot g_2(x_i))$, $\sum_{\oplus} \in PA$, $\{g_1, g_2\} \in PS^{\circ}$. Then, we define the following two splitting rules (SR):

SR1: $\sum(g_1(x_i) \odot g_2(x_i)) = \sum(g_1(x_i)) \odot \sum(g_2(x_i)), \odot \in \{+, -\}$;
SR2: $\prod(g_1(x_i) \odot g_2(x_i)) = \prod(g_1(x_i)) \odot \prod(g_2(x_i)), \odot \in \{\times, /\}$.

By applying the above two rules, aggregation states in $(PS^{\odot} \setminus PS^{\circ})$ can be split into new ones with scalar functions in PS°, which can still be verified using Theorem 4.1. If aggregation stares are not covered by splitting rules in this case, SUDAF simply proceeds syntactic comparison between their mathematical expressions. Note that syntactic comparison is sufficient but not necessary.

5 A PRACTICAL APPROACH TO SOLVE THE SHARING PROBLEM

We present in this section a practical approach to solve the sharing problem based on the results provided by Theorem 4.1. Turning the conditions of Theorem 4.1 into an algorithm could be cumbersome because equivalent mathematical expressions may have different syntactic shapes.

Example 5.1. Consider the problem whether $s_1(X) = \sum 4x_i^2$ shares $s_2(X) = \sum(3x_i)^2$. Using Theorem 4.1, one needs to construct $f_1 \circ f_2^{-1}(x) = 4x \circ x^2 \circ \frac{1}{3}x \circ \sqrt{x}$ (over the positive domain since both f_1 and f_2 are even). Then, according to case 2.1 of Theorem 4.1, we need to check whether $f_1 \circ f_2^{-1}(x) = ax$, for some constant a. This is not an easy task, particularly for general cases, since it requires mathematical transformations of the original expression as follows: $f_1 \circ f_2^{-1}(x) = 4x \circ x^2 \circ \frac{1}{3}x \circ \sqrt{x} = 4x \circ \frac{1}{9}x \circ x^2 \circ \sqrt{x} = \frac{4}{9}x$. The first transformation is a *reordering* of $x^2 \circ \frac{1}{3}x$, which generates $\frac{1}{9}x \circ x^2$, and it is then followed by a *removal* of the composition $x^2 \circ \sqrt{x}$. Finally, $f_1 \circ f_2^{-1}(x)$ is transformed to $\frac{4}{9}x$, which satisfies the condition $f_1 \circ f_2^{-1}(x) = ax$, with $a = \frac{4}{9}$, of the case 2.1 of Theorem 4.1.

In addition, a straightforward implementation of Theorem 4.1 leads to redundant computations as illustrated below.

Example 5.2. Checking whether $s'_1 = \sum 6x_i^3$ shares $s'_2 = \sum(5x_i)^3$ requires redoing identical transformations as in the previous example (i.e., checking whether $s_1(X) = \sum 4x_i^2$ shares $s_2(X) = \sum(3x_i)^2$). This is because we have as a general property: $\sum a_2 x_i^{a_1}$ shares $\sum(b_1 x_i)^{b_2}$ if $a_1 = b_2$.

Hence, our general idea to deal with the two previous issues is: (i) to use symbolic representations of aggregation states to avoid redundant computations, i.e., using $\sum a_2 x_i^{a_1}$ and $\sum(b_1 x_i)^{b_2}$, where a_1, a_2, b_1 and b_2 are parameters, to represent the *concrete* states $\sum 4x_i^2$ and $\sum(3x_i)^2$, and (ii) to precompute sharing relationships between symbolic representations to avoid cumbersome transformations of mathematical expressions at execution time. For example, we precompute the relationship stating that $\sum a_2 x_i^{a_1}$ shares $\sum(b_1 x_i)^{b_2}$ if $a_1 = b_2$. Then, at execution time, this relationship can be used to efficiently identify that the *concrete* aggregation state $\sum 4x_i^2$, an instance of the abstract state $\sum a_2 x_i^{a_1}$, shares the concrete state $\sum(3x_i)^2$, an instance of the abstract state $\sum(b_1 x_i)^{b_2}$, because the condition $a_1 = b_2$ is satisfied.

5.1 Symbolic representations

In this section, we first present symbolic representations of scalar functions and then use them to introduce symbolic representations of aggregation states. In the sequel, we assume an infinite set of parameters, distinct from the set of constants. Hereafter, the parameters are denoted $p, p_1, \ldots$.

Symbolic primitive scalar functions. Intuitively, px with a parameter p is the symbolic representation of the primitive scalar function $2x$. In this case, $2x$ is an instance of px. Formally, we consider four symbolic primitive scalar functions with a parameter p: $px = \{ax | \forall a \neq 0\}$; $log_p x = \{log_a x | \forall a > 0, \neq 1\}$; $p^x = \{a^x | \forall a > 0, \neq 1\}$; $x^p = \{x^a | \forall a \neq 0\}$. We use the notation $sf_{\bar{p}}(x)$ for a symbolic primitive scalar function with a sequence $\bar{p} = (p)$ of a parameter p.

Symbolic scalar functions. Intuitively, $p_2 x^{p_1}$ with a parameter sequence (p_2, p_1) is the symbolic representation of the scalar function $3x^2$, and in this case $3x^2$ is an instance of $p_2 x^{p_1}$. Formally, let every $sf_{i\bar{p}_i}(x)$ for $i \in [1, \ldots, l]$ be a symbolic primitive scalar function. Then, $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ \ldots \circ sf_{1\bar{p}_1}(x)$ is a symbolic scalar function $sf_{\bar{p}}(x)$ with a sequence $\bar{p} = (p_l, \ldots, p_1)$ of parameters. Similarly, $|sf_{\bar{p}}| = l$.

Symbolic aggregation states. Intuitively, $\sum p_2 x_i^{p_1}$ is the symbolic representation of $\sum 3x_i^2$. In this case, $\sum p_2 x_i^{p_1}$ is called a symbolic (aggregation) state and we say that the concrete state $\sum 3x_i^2$ is an instance of the symbolic state $\sum p_2 x_i^{p_1}$. Formally, let $\sum_{\oplus} \in PA$ and $sf_{\bar{p}}(x)$ be a symbolic scalar function. Then, $ss(X) = \sum_{\oplus} sf_{\bar{p}}(x_i)$ is a symbolic aggregation state.

Specifically, we let $\sum x_i$ and $\prod x_i$ be also two symbolic aggregation states, which contain respectively only one instance $\sum x_i$ and $\prod x_i$, and we define $|f| = 0$ with $f(x) = x$.

5.2 Precomputed sharing relationships

Informally, we say that a symbolic state ss_1 shares a symbolic state ss_2 if and only if for any instance s_1 of ss_1, there exists an instance s_2 of ss_2, such that s_1 shares s_2. As explained previously, our aim is to precompute and store the sharing relationships between symbolic aggregation states. Specifically, we conduct an exhaustive analysis to identify the sharing relationships between symbolic states in a preprocessing step, which is performed once when SUDAF is deployed, and then the precomputed relationships are reused at runtime to handle the sharing problem between concrete aggregation states. Note that the space of symbolic states may be very huge (theoretically infinite) because symbolic scalar functions may be of arbitrary lengths. In addition, aggregation states having scalar functions with a higher length are useless from the practical point of view. For example in our experiments presented in Section 6 it was enough to use aggregation states, whose scalar functions have a length up to 2, to express aggregations in real-world applications. Therefore, SUDAF enables a user to bound the space of symbolic aggregation

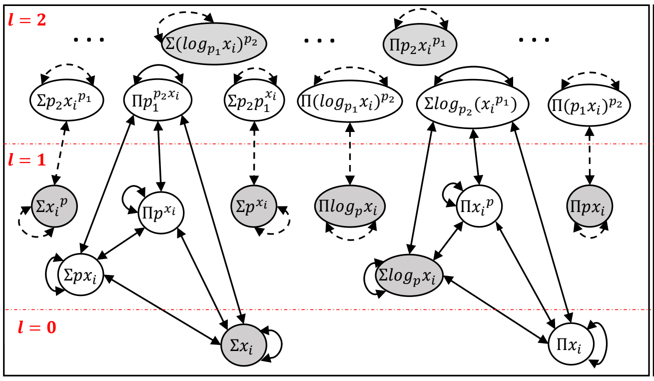

Figure 4: The digraph G of $saggs_2(X)$.

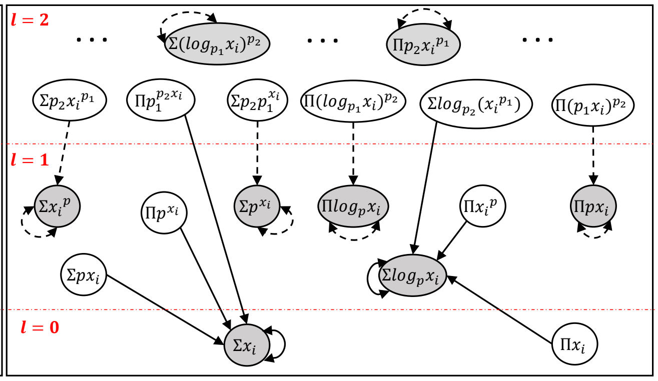

Figure 5: The simplified digraph G of $saggs_2(X)$.

states that is prebuilt in the preprocessing step using a configuration parameter, denoted by l. The obtained space, denoted by $saggs_l(X)$, is introduced below.

l-bounded symbolic space. Let $l \geqslant 0$ be an integer. We define the space $saggs_l(X)$ of symbolic aggregation states as follows: $saggs_l(X) = \{\sum_\oplus sf_{\bar{p}}(x_i) | sf_{\bar{p}}$ is a symbolic scalar function with $|sf_{\bar{p}}| \leqslant l\}$. We say $saggs_l(X)$ is a l-bounded symbolic space. Note that the size of the set $saggs_l(X)$ is bounded by $\frac{2(4^{l+1}-1)}{3}$.

Once the parameter l is fixed by a user, SUDAF builds space $saggs_l(X)$ and precomputes the sharing relationships between every two symbolic aggregation states in $saggs_l(X)$. An excerpt of $saggs_2(X)$ is shown in Figure 4, where each symbolic aggregation state is depicted as a node labeled with its expression (the meaning of edges in Figure 4 is explained later). As it can be observed in Figure 4, the space $saggs_2(X)$ is organized in three levels, where each level i, with $i \in \{0, 1, 2\}$, contains the symbolic states of the form $\sum_\oplus sf_{\bar{p}}(x_i)$ with $|sf_{\bar{p}}| = i$. Figure 4 shows all the symbolic states of level 0 and 1, and some states of level 2.

5.3 Organizing the space $saggs_l(X)$

We briefly discuss the organization of $saggs_l(X)$, w.l.o.g., focusing on the case $l = 2$. In the sequel, we first consider that the input multiset X contains only positive values, i.e., $X \in \mathcal{M}(\mathbb{Q}_+)$, then we extend the results to the case where X contains both negative and positive values. We represent the sharing relationships between symbolic states in $saggs_2(X)$ using a digraph $G = (V, E)$ where the set of vertices $V = saggs_2(X)$ is the space $saggs_2(X)$ and the set of edges $E \subseteq V \times V$ represent the sharing relationship, i.e., $(ss', ss) \in E$ if and only if ss' shares ss. Figure 4 depicts the digraph associated with the space $saggs_2(X)$. We distinguish between two kinds of sharing relationships in G (two types of edges are depicted in Figure 4). The first one is called *strong relationships* and relates two symbolic states (ss', ss) if ss' shares ss without requiring any condition on the parameters. The second one is called *weak relationships* and relates two symbolic states (ss', ss) if ss' shares ss under some conditions defined over the parameters of ss and ss'. For example, since any instance of $\sum px_i$ shares any instance of $\prod p^{x_i}$, then $\sum px_i$ and $\prod p^{x_i}$ have a strong sharing relationship denoted as $\sum px_i \rightarrow \prod p^{x_i}$. As another example, the state $\sum x_i^p$ shares $\sum p_2 x^{p_1}$ with the condition $p = p_1$, then $\sum x_i^p$ and $\sum p_2 x^{p_1}$ have a weak sharing relationship denoted as $\sum x_i^p \overset{p=p_1}{\dashrightarrow} \sum p_2 x^{p_1}$.

We observed that in the space $saggs_2(X)$, the sharing relationships are *equivalence relations*. For example, $\sum px_i \leftrightarrow \prod p^{x_i}$ and $\sum x_i^p \overset{p=p_1}{\leftarrow\!\!\rightarrow} \sum p_2 x^{p_1}$. Consequently, the space $saggs_2(X)$ can be partitioned into *equivalence classes*. Intuitively, for a symbolic state ss, its associated equivalence class, denoted $[ss]$, is made of the set of symbolic aggregation states that shares (and are shared by) ss. For example, as depicted in Figure 4: $[\sum x_i] = \{\sum x_i, \sum px_i, \prod p^{x_i}, \prod p_1^{p_2 x_i}\}$ and $[\sum x_i^p] = \{\sum x_i^p, \sum p_2 x_i^{p_1}\}$.

We select a unique element in each equivalence class $[ss]$ to be a *representative* of the class, which is denoted as $rep([ss])$ and depicted as a shaded node in Figure 4. It is clear that, given an equivalence class $[ss]$, one only needs to focus on the instances of its representative $rep([ss])$ since they are able to compute an instance of any other element in $[ss]$.

We simplify G presented in Figure 5 based on the equivalence relations derived from the sharing relationships. More precisely, it is only necessary for any state $ss \in saggs_2(X)$ to store such a sharing relationship $ss \rightarrow rep([ss])$, or $ss \overset{pcon}{\dashrightarrow} rep([ss])$ with a parameter condition ($pcon$). Consequently, when an instance s of ss is given, we use an edge $ss \rightarrow rep([ss])$, or $ss \overset{pcon}{\dashrightarrow} rep([ss])$ to get a cached instance of $rep([ss])$ to compute s.

Extension to an arbitrary multiset. When a multiset X contains negative values, instances of some symbolic states in $saggs_2(X)$ do not exist, which will cause the miss of sharing opportunities. We take $\sum log_p x_i$ as an example to explain the issue. As we know that, an instance $\sum ln(x_i)$ of $\sum log_p x_i$ can only be computed over the positive domain, such that the caches for $\sum log_p x_i$ are empty in this context. To deal with this issue, we separate input values from their signs. Specifically, we translate an input multiset $X = \{x_1, \dots, x_n\}$ to the following multiset $\hat{X} = \{(|x_1|, sgn(x_1)), \dots, (|x_n|, sgn(x_n))\}$, where $|x_j|$ denotes the absolute value of x_j and $sgn(x_j)$ is its sign. Then, we keep in the cache such a result $(\sum ln|x_i|, \prod sgn(x_i))$ for $\sum log_p x_i$. By this way, a new aggregation state $\sum ln(x_i^2)$ can still be computed using the cache $(\sum ln|x_i|, \prod sgn(x_i))$ that is stored for $\sum log_p x_i$.

6 EXPERIMENTAL EVALUATION

We implemented a SUDAF prototype in Java and Scala, which can be used on top of PostgreSQL (through JDBC) and Spark SQL. The SUDAF prototype also comes equipped with a UDAF editor that enables users to write SUDAF-compatible UDAFs and integrate them in SQL queries.

The general scheme of our experiments is the following. We select 3 query models, and we instantiate each query model using 11 aggregations. We simulate the 11 instances of each query model coming in 2 different orders, i.e., two different sequences of

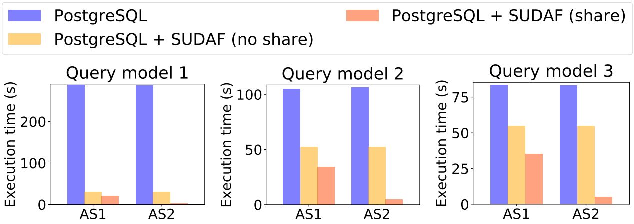

Figure 6: Total execution time of each query sequence in each query model.

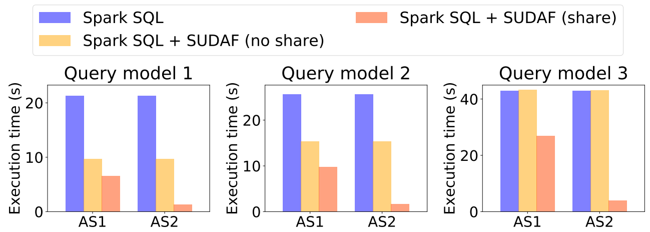

Figure 7: Total execution time of each query sequence in each query model.

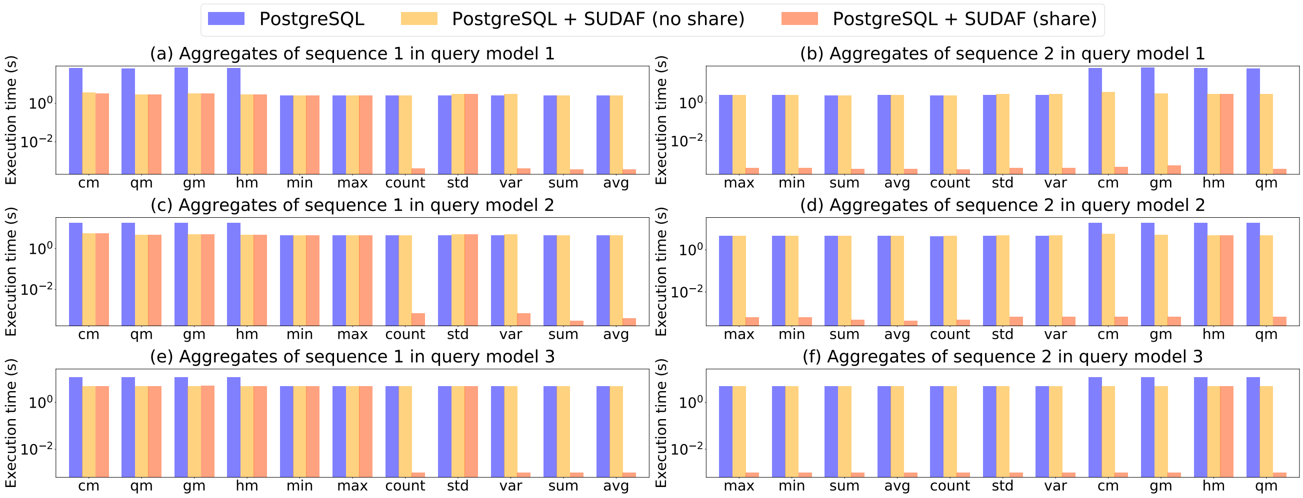

Figure 8: Execution time in PostgreSQL of each query in each query sequence.

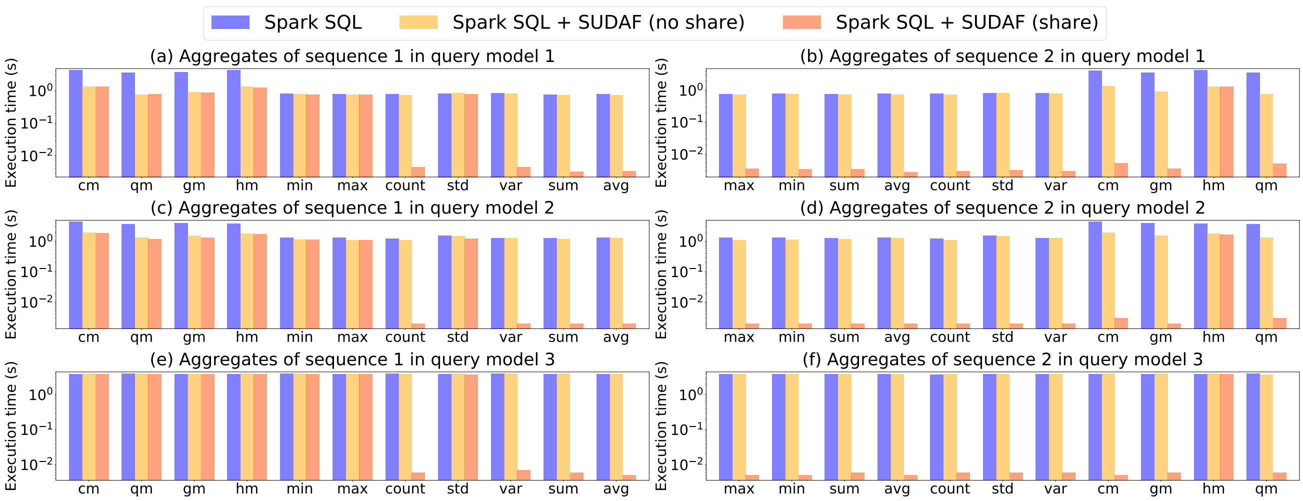

Figure 9: Execution time in Spark SQL of each query in each query sequence.

queries. Thus, the tested workload consists of 6 query sequences, where each sequence has 11 queries. We execute the query sequences in three technical contexts (i) PostgreSQL and Spark SQL, (ii) SUDAF without the sharing functionality, and (iii) SUDAF with the sharing functionality. In the PostgreSQL environment (case (i)), the aggregations are either PostgreSQL built-in or hard-coded user-defined functions, and similarly for the Spark SQL environment. PostgreSQL UDAFs are created using PL/pgSQL, and Spark SQL UDAFs are created using the UserDefinedAggregateFunction interface in Scala code. In the SUDAF environment (cases (ii) and (iii)), UDAFs are provided as mathematical expressions and used in the SQL queries. And in case (iii) of SUDAF, the precomputed sharing relationships in $saggs_2(X)$ are exploited to reuse cached aggregation states to compute new ones. In SUDAF sharing environment, we prefetch a moment sketch (MS) [16, 26] under one of the two selected query orders. At the end of this section, we also present a scenario of running a random sequence of 200 queries in the Spark SQL context.

Our main findings are twofold. First, we observed that SUDAF without sharing outperforms both PostgreSQL and Spark SQL despite the overhead in SUDAF due to the analysis and decomposition of UDAF expressions. The main reason that explains these performances comes from the fact that rewriting of UDAFs by SUDAF, which is based on canonical forms, leads to implementations that use PostgreSQL or Spark SQL built-in functions, these later ones being much faster than PostgreSQL or Spark SQL UDAFs. The second finding is SUDAF with sharing outperforms both PostgreSQL and Spark SQL. In particular, the fine-grained unit of caching used in SUDAF improves the sharing possibilities and increases the gain brought by sharing.

Experiment setup. All experiments of Spark SQL are performed on a cluster with 1 master node and 6 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 on PostgreSQL are only performed on the master node running PostgreSQL 11.4.

Query models. The three query models used in experiments are illustrated below, where AGG represents an aggregation.

```
-- Query model 1
SELECT AGG(internet_traffic) FROM milan_data;
-- Query model 2
SELECT square_id, AGG(internet_traffic) FROM milan_data
GROUP by square_id ORDER by square_id LIMIT 20;
-- Query model 3, the TPC-DS query 7 when AGG is avg
SELECT i_item_id, AGG(ss_quantity) agg1, AGG(ss_list_price) agg2,
       AGG(ss_coupon_amt) agg3, AGG(ss_sales_price) agg4
FROM   store_sales, customer_demographics, date_dim, item, promotion
WHERE  ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and cd_gender = 'M'
       and cd_marital_status = 'S' and
       cd_education_status = 'College' and
       (p_channel_email = 'N' or p_channel_event = 'N')
       and d_year = 2000
GROUP BY i_item_id ORDER BY i_item_id LIMIT 100;
```

Datasets. The first two query models are evaluated on the Milan dataset [22] and the third query model is evaluated on the TPC-DS [27] dataset. For the experiments of PostgreSQL, the Milan dataset consists of 72.6 million rows in total and the TPC-DS dataset comes with scale = 20. For the experiments of Spark SQL, the Milan dataset consists of 319 million rows in total and the TPC-DS dataset comes with scale = 100. All data files in Spark SQL experiments are in Parquet format.

Aggregate functions. We use the following 11 aggregate functions to instantiate our query models: cubic_mean (cm), quadratic_mean (qm), geometric_mean (gm), harmonic_mean (hm), min, max, count, sum, average (avg), standard deviation (std), variance (var). In the used PostgreSQL and Spark SQL version, all of these functions are built-in functions except the functions cm, qm, gm and hm which are implemented using PL/pgSQL in PostgreSQL and using *UserDefinedAggregateFunction* interface in Scala code in Spark SQL.

Query sequences. We instantiate each query model using each of the 11 aggregations and define the following two sequences of query executions for each instantiated query model:

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 obtain 6 query sequences in total, where each query sequence is made of 11 aggregate queries. In the SUDAF sharing environment (cases (ii)) with the sequence AS2, we prefetch a moment sketch (MS) [16, 26] with parameter k = 10, which consists of a set of aggregate functions ($min, max, count, \sum x_i, ..., \sum x_i^k, \sum ln(x_i), ... \sum ln^k(x_i)$) and can be used to approximate a percentile, e.g., median.

Experimental results. We executed the 6 query sequences on PostgreSQL or Spark SQL, SUDAF without sharing, and SUDAF with sharing, and we report the execution time of every query. In scenarios with sharing, we use precomputed sharing relationships of symbolic aggregation states in $saggs_2(X)$, and we also add three additional relationships for SQL standard aggregates, max, min, and count, that they share themselves. Note that in the reported results we do not take into account the overhead needed to precompute sharing relationships in $saggs_2(X)$ which is part of the initialization of SUDAF and takes 110 ms. However, the overhead due to the cache access is included in the global execution time reported for each query. This overhead is about 2 ms for query model 1 or 2, and about 5 ms for query model 3. Moreover, the prefetching of a moment sketch is a preprocessing step in the aggregate sequence AS2, and the corresponding time is not taken into account. In the context of PostgreSQL, the prefetching time is 13.06 s for query model 1, 15.16 s for query model 2, and 14.53 s for query model 3. In the context of Spark SQL, the prefetching time is 1.87 s for query model 1, 2.17 s for query model 2, and 3.82 s for query model 3.

The total execution time of each query sequence in each query model is presented in Figure 6 for the case of PostgreSQL and in Figure 7 for the case of Spark SQL. We observe that PostgreSQL or Spark SQL (respectively, SUDAF without sharing) always have the same execution time for the two sequences of the same model. Also, we observe that SUDAF without sharing outperforms both PostgreSQL and Spark SQL in all the considered scenarios except query model 3 in Spark SQL (the reason is explained later). SUDAF with sharing shows the best performances, whatever the considered sequence or query model. In the sequel, we discuss the execution time of every individual query depicted in Figure 8 and 9 for the cases of PostgreSQL and Spark SQL.

SUDAF without sharing. In this scenario, SUDAF only rewrites aggregations to built-in ones and it does not share computations in processing query sequences. For the case of PostgreSQL, compared to PostgreSQL UDAF queries, SUDAF speeds up UDAF queries up to 20X in query model 1 (Figure 8 (a) and (b)), 4X in query model 2 (Figure 8 (c) and (d)), and 2X in query model 3 (Figure 8 (e) and (f)). For the case of Spark SQL, compared to Spark UDAF queries, SUDAF speeds up UDAF queries up to 3X in query model 1 (Figure 9 (a) and (b)), 2X in query model 2 (Figure 9 (c) and (d)), and have identical query time in query model 3 (Figure 9 (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 PostgreSQL or Spark SQL built-in functions, which are faster compared to PostgreSQL or Spark UDAFs. The performance improvements of such a rewriting depends on the number of data to be aggregated. The instances of query model 1 have the highest number of values to be aggregated while the instances of query model 3 have the smallest number of values as aggregation inputs. Therefore, for the case of query model 3, the difference between SUDAF only with the rewriting functionality and Spark SQL is less noticeable.

SUDAF with sharing. In this scenario, SUDAF rewrites aggregations to built-in ones and shares the computation results of partial aggregations in every query sequence. For the sequence AS1, we observe in Figure 8 (a), (c) and (e) and in Figure 9 (a), (c) and (e) that for all the considered query models the computation times of count, variance (var), sum and average (avg) decrease drastically w.r.t. the no sharing option. This is because SUDAF

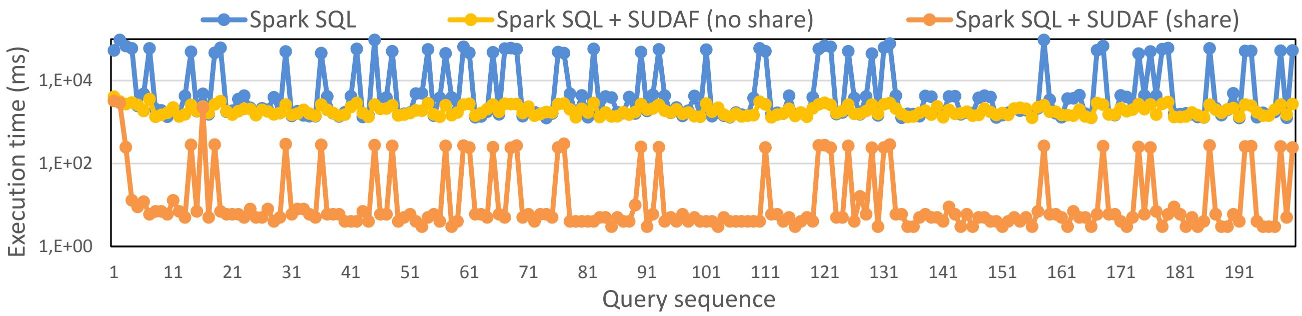

Figure 10: Execution time in Spark SQL of a random sequence of 200 queries.

is able to reuse cached results from earlier aggregates in the sequence AS1. As it can be observed in Figure 8 (b), (d) and (f) and in Figure 9 (b), (d) and (f), the sequence AS2 is more advantageous for sharing due to the prefetched moment sketch. Indeed, the moments sketch consists of 33 partial aggregates which are cached by SUDAF and reused for the computation of all the remaining aggregations in the sequence AS2 except the harmonic mean (hm). Computing queries with the harmonic mean in AS2 still requires data access since the aggregation state $\sum x_i^{-1}$ in the harmonic mean is not evaluated in previous computing.

Random query sequence. We present in Figure 10 the scenario of running a random sequence of 200 queries in Spark SQL, which are instances of the query model 2 having the following 16 aggregate functions: (min, max, sum, avg, harmonic_mean, quadratic_mean, cubic_mean, geometric_mean, stddev, variance, skewness, kurtosis, approx_median, count, approx_first_quantile, approx_thrid_quantile). The benefits of using SUDAF in this scenario are more obvious (the orange line in Figure 10).

7 RELATED WORKS

There is a wealth of research on queries with aggregations, earlier works focusing on standard aggregations (e.g., [8, 9, 12, 18, 19, 35]) and then extended to UDAFs (e.g., [6, 10, 20, 24]). Partial aggregation appeared as an essential technique used to improve the performances of aggregations: instead of computing aggregations on a complete multiset, applying aggregations on subsets and merging intermediate results is an efficient solution in numerous scenarios. In OLAP applications, partial aggregation enables computing aggregation by merging summaries of cells with different granularities across multi-dimensional data, thereby allowing aggregate queries to be executed on pre-computed results instead of base data [8]. In join-aggregate query optimization, partial aggregation enables to compute group-by aggregation before joins to decrease the size of intermediate results [35], i.e., the eager group-by technique. In distributed computing, partial aggregation allows to push the execution of aggregation before transferring data on networks [36], thereby decreasing the overhead of data shuffling, which is usually called initial reduce in MapReduce-like frameworks. An original classification of aggregations [18] distinguishes between algebraic aggregations having partial aggregation with fixed size results, and holistic functions where there is no constant bound on the storage size for partial aggregation. Several properties are proposed to have partial aggregations from algebraic aggregations, such as decomposable aggregation [35], commutative semi-group aggregation [11] and associative and commutative aggregation [36].

Most modern data management and analysis systems support UDAFs (e.g., [1, 2, 4, 21, 28, 29]). In the original MapReduce (*MR*) framework [3, 14], UDAFs are implemented according to the *MR* paradigm without requiring any specific template. This makes the semantics of UDAFs hidden in the implementations and hinders optimization possibilities (e.g., reordering with relational operators and other UDAFs [20]). However, in most of recent systems, users define UDAFs using an *IUME* pattern (initialize function, update function, merge function and evaluate function). Although such an approach enables exploiting the properties of the merging functions to allow optimization based on partial aggregations, e.g., parallel computation of the merging functions, part of the UDAF semantics still remains hidden in the implementation, which hampers the opportunity of aggregate sharing. In addition, implementing UDAFs in existing frameworks may be a tedious task since it is up to the user to map a UDAF to the implementation paradigm (*MR* or *IUME*). We build on a canonical form of UDAFs proposed in [10] to design SUDAF by allowing users to specify UDAFs as mathematical expressions and then automatically generate canonical forms of UDAFs which are compliant with the $IUME$ pattern. Consequently, with SUDAF a user does not need to handle the problem of how to obtain partial aggregations from UDAFs. Moreover, SUDAF knows the semantics of partial aggregations (primitive functions used in partial aggregation) which extends the optimization opportunities.

Different facets of the sharing problem have been studied in the literature, e.g., rewriting aggregate queries using materialized views [11, 12], reusing caches to accelerate multi-dimensional queries [8, 15], or identifying overlapping processing for multiple aggregate queries with various selection predicates [19], group-by attributes [9] and sliding-windows [5, 23]. Most of these approaches focus on the data dimension, i.e., they consider the problem of sharing the same aggregation across different ranges or granularities of data. Our work does not consider the data granularity dimension where existing techniques, e.g., [15, 33], can be used to extend SUDAF in this direction. [10, 11] proposes to predefine computation rules for sharing between different aggregations. However, SUDAF automatically identifies sharing opportunities on partial aggregates across different UDAFs.

The closest work to SUDAF is DataCanopy [33]. DataCanopy caches the basic aggregates (e.g., $\sum x_i$, $\sum x_i^2$ and $\sum x_i y_i$) of statistical measures and then is able to reuse them for queries with various range predicates. Basic aggregates are maintained at a granularity of a chunk (smallest portion of data), and DataCanopy allows sharing across queries covering overlapping chunks. In DataCanopy, basic aggregates are fixed in advance and the decomposition of an aggregate into basic ones is predefined (see Table 1 of [33]). We discuss the differences between DataCanopy and SUDAF as follows. From a theoretical standpoint, the sharing condition in SUDAF allows having a scalar function between two aggregates (see Theorem 4.1), which is more general compared to sharing identical basic aggregates in DataCanopy. From a practical standpoint, our approach is complementary to DataCanopy in the sense that DataCanopy deals with sharing w.r.t. the data dimension and proposes a static approach for sharing on the aggregation dimension, whereas SUDAF extends its static approach to a dynamic one w.r.t. the aggregation dimension. More precisely, the sharing opportunities w.r.t the aggregation dimension are automatically identified in SUDAF, which do not require any decomposition rule and are not restricted to a fixed set of aggregates. For example, if we restrict the attention to the set of predefined basic aggregates introduced in [33], the execution of a geometric mean ($gm(X) = exp(\frac{\sum ln(x_i)}{count}, \forall x_i > 0)$ cannot take any benefit from the static caching solution used in DataCanopy (i.e., cannot reuse the basic aggregates stored in the cache and do not lead to any new cached computation results). In contrast, SUDAF can reuse partial aggregates from the cache to compute *gm* and if not possible, it caches the partial aggregates $(\sum ln(x_i), count)$ after computing *gm* from base data. To obtain similar behavior, one needs to explicitly define additional basic aggregates in DataCanopy together with the appropriate decomposition rules for *gm*. In addition to being cumbersome, such a task requires to know in advance the query workloads that will be issued.

8 CONCLUSIONS AND FUTURE WORKS

In this paper, we introduce the design principles underlying SUDAF, a framework that provides a set of primitive functions together with a composition operator to enable users to define mathematical expressions of their UDAFs. SUDAF comes equipped with the ability to automatically rewrite partial aggregations, which are factored out from mathematical expressions of UDAFs, using built-in aggregates, and supports efficient dynamic caching and sharing of partial aggregates. We showed experimentally the benefits of rewriting partial aggregates of UDAFs using built-in functions and sharing partial aggregates to improve the performances of queries with UDAFs.

In this paper, we focus on the issue of *how to compute a UDAF from another UDAF*. In practice, to share computation results of different queries, we need to consider the data dimension, e.g., different range queries, or different OLAP queries. Sharing over data dimension has been extensively studied in existing works [15, 33]. The general idea is to split cached query results using chunks. For the case of range queries, a chunk is a range predicate over an attribute. For the case of OLAP queries, a chunk is a region in a multi-dimensional space. Merging our sharing approach with such approaches, we can share computation results for different queries with different UDAFs. As another future work, we envision to exploit the fact that the semantics of UDAFs is known by SUDAF to investigate query optimization and query rewriting problems for join and group-by queries with UDAFs.

9 ACKNOWLEDGMENTS

This research was supported by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25) and by the CPER of the "Région Auvergne-Rhône-Alpes", the French Government and FEDER from the European community.

REFERENCES

[1] Aache Hive. 2019. https://hive.apache.org.
[2] Apache Flink. 2019. https://flink.apache.org.
[3] Apache Hadoop. 2019. https://hadoop.apache.org.
[4] Apache Spark. 2019. https://spark.apache.org/.
[5] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-window Aggregates *(VLDB '04)*. VLDB Endowment, 336–347.
[6] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. 1201–1210.
[7] Central moments. 2019. https://en.wikipedia.org/wiki/Central_moment.
[8] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.* 26, 1 (March 1997), 65–74.
[9] Zhimin Chen and Vivek Narasayya. 2005. Efficient Computation of Multiple Group by Queries. In *SIGMOD '05*. ACM, New York, NY, USA, 263–274.
[10] Sara Cohen. 2006. User-defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD '06*. ACM, New York, NY, USA, 49–60.
[11] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2006. Rewriting Queries with Arbitrary Aggregation Functions Using Views. *ACM Trans. Database Syst.* 31, 2 (June 2006), 672–715.
[12] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 1999. Rewriting Aggregate Queries Using Views. In *PODS '99*. ACM, New York, NY, USA, 155–166.
[13] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 2000. Algorithms for Rewriting Aggregate Queries Using Views. In *ADBIS-DASFAA '00*. Springer-Verlag, London, UK, UK, 65–78.
[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*. San Francisco, CA, 137–150.
[15] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. In *SIGMOD '98*. ACM, New York, NY, USA, 259–270.
[16] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1647–1660.
[17] Michel Grabisch, Jean-Luc Marichal, Radko Mesiar, and Endre Pap. 2011. Aggregation function: Means. *Information Sciences* 181, 1 (January 2011), 1–22.
[18] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery* 1, 1 (01 Mar 1997), 29–53.
[19] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. 2007. Sharing Aggregate Computation for Distributed Queries. In *SIGMOD '07*. ACM, New York, NY, USA, 485–496.
[20] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. *ICDE*.
[21] IBM DB2. 2019. https://www.ibm.com/analytics/db2.
[22] Telecom Italia. 2015. Telecommunications - SMS, Call, Internet - MI. https://doi.org/10.7910/DVN/EGZHFV
[23] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly Sharing for Streamed Aggregation. In *SIGMOD '06*. 623–634.
[24] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD '17*. 1717–1722.
[25] Radko Mesiar Michel Grabisch, Jean-Luc Marichal and Endre Pap. 2009. *Aggregation Functions*. Cambridge University Press, Cambridge.
[26] Moment-based quantile sketches for aggregations. 2018. https://github.com/stanford-futuredata/msketch.
[27] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB '06*. 1049–1058.
[28] Oracle. 2019. https://docs.oracle.com/.
[29] PostgreSQL. 2019. https://www.postgresql.org/docs/.
[30] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD '16*. 3–18.
[31] Sharing computations for user-defined aggregate functions (technical report). 2019. https://github.com/CHAOHIT/SUDAF/blob/master/sudaf-technical-report.pdf.
[32] Standardized moments. 2019. https://en.wikipedia.org/wiki/Standardized_moment.
[33] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data Canopy: Accelerating Exploratory Statistical Analysis. In *SIGMOD '17*. ACM, New York, NY, USA, 557–572.
[34] Wolfram Mathematica. 2019. https://reference.wolfram.com/language/guide/MathematicalFunctions.
[35] Weipeng P. Yan and Per-Ake Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB '95*. 345–357.
[36] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *SOSP '09*. ACM, New York, NY, USA, 247–260.