Optimization of User-Defined Aggregate Functions : Parallization and Sharing

Chao Zhang

UNIVERSITÉ CLERMONT AUVERGNE

ECOLE DOCTORALE DES SCIENCES POUR L'INGÉNIEUR

DOCTORAL THESIS

Optimization of User-Defined Aggregate Functions: Parallelization and Sharing

---

A thesis submitted in fulfillment of the requirements
for the degree of Docteur of Université Clermont Auvergne
25th November 2019

Author:
Chao ZHANG

Supervisor:
Farouk TOUMANI
Emmanuel GANGLER

Jury:

M. Dimitris Kotzinos,	Professor	*Reviewer*
M. Reza Akbarinia,	Research Director	*Reviewer*
Mme Angela Bonifati,	Professor	*Examiner*
Mme KANG - Myoung-Ah,	Associate Professor	*Examiner*
M. Farouk Toumani,	Professor	*Supervisor*
M. Emmanuel Gangler,	Research Director	*Supervisor*

LIMOS, UMR-6158, CNRS
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes
Clermont-Ferrand, France

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
LIMOS, UMR-6158, CNRS
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes
Clermont-Ferrand, France

Abstract

Optimization of User-Defined Aggregate Functions: Parallelization and Sharing

by Chao ZHANG

Applications of aggregations for information summary have great meanings in various fields. System built-in aggregations are not sufficient to cover the needs of new applications in the age of analytics. 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 UDAFs requires hardcoding the routine that computes the aggregation function, and the semantics of UDAFs is totally or partially unknown to the query processor which hampers the optimization possibilities. This thesis presents 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 comes equipped with the ability to generate efficient parallel implementation from users' UDAFs automatically and supports dynamic caching and reusing of partial aggregates. Our experiments show that the proposed sharing technique can lead from one to two orders of magnitude improvement in query execution times.

Partial aggregations, Query optimization, Query rewriting, User-defined aggregate functions, MapReduce

UNIVERSITÉ CLERMONT AUVERGNE
Ecole Doctorale des Sciences Pour l'Ingénieur
LIMOS, UMR-6158, CNRS
Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes
Clermont-Ferrand, France

Résumé

Optimisation des fonctions d'agrégation définies par l'utilisateur: parallélisation et partage

par Chao ZHANG

Les applications des agrégations pour la synthèse d'informations sont significatives dans de nombreux domaines. Les agrégations incorporées par défaut dans les systèmes ne sont pas suffisantes pour satisfaire les besoins qui émergent avec les progrès de l'analyse de données. Les UDAFs (User-Defined Aggregate Functions ou, en français, fonctions d'agrégation définies par l'utilisateur) sont en train de devenir un des opérateurs fondamentaux en analyse de données avancée. Le mécanisme UDAF fourni par la plupart des systèmes modernes souffre cependant d'au moins deux défauts : la définition d'UDAFs nécessite le codage en dur de la routine qui calcule la fonction d'agrégation, et la sémantique des UDAFs est totalement ou partiellement inconnue des processeurs de requêtes, empêchant leur optimisation. Cette thèse présente SUDAF (Sharing User-Defined Aggregate Functions), un cadre framework déclaratif qui permet aux utilisateurs de formuler des UDAFs sous la forme d'expressions mathématiques et de les utiliser dans des déclarations SQL. SUDAF est capable de générer automatiquement des implémentations parallèles efficientes à partir des UDAFs des utilisateurs, et supporte la mise en cache dynamique et la réutilisation des agrégats partiels. Nos expérimentations montrent que la technique de partage proposée permet des gains d'un à deux ordres de magnitude sur les temps d'exécution des requêtes.

Partial aggregations, Query optimization, Query rewriting, User-defined aggregate functions, MapReduce

For my parents and grandparents.

Acknowledgements

This work is fully funded by University Clermont Auvergne and the department of Puy-de-Dôme. I appreciate having the opportunity to work on this thesis.

I would like to express my sincere gratitude to my advisor Prof. Farouk Toumani for his motivation and immense knowledge. I also would like to thank my advisor Dr. Emmanuel Gangler for his continuous support of this work. Besides my advisors, I would like to thank the rest of my thesis committee: Dr. Reza Akbarinia, Prof. Dimitris Kotzinos, Prof. Angela Bonifati, and Prof. KANG - Myoung-Ah, for their insightful comments and valuable suggestions.

Last but not least, I would like to thank my parents and grandparents for supporting me spiritually and thank my friends for preparing the defense.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An aggregate function has the inherent property of taking several values as input and generating a single value based on specific criteria [MGP09, GMMP11]. This ability to summarize information, the intrinsic feature of aggregation, has always been a fundamental task in data analysis [GCoS+97, KBY17]. While earlier data management and analytical 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., [apaa, apad, apab, RDBc, RDBa, RDBb]) enable users to extend the system functionalities by defining their own aggregations. The UDAF (User-Defined Aggregate Function) mechanism provides a flexible interface to define new aggregate functions that can then be used for advanced data analytics, i.e., queries with statistical functions or ML workloads.

Efficiently computing aggregate functions (built-in functions or UDAFs) is essential to processing queries with aggregations. In the age of massively distributed and parallel computing, aggregate functions have to be decomposed into a partial aggregation, a merging function, and a finalizing function. Partial aggregations are sent to compute at worker nodes, results of which are collected at a master node. Then, the merging function and finalizing function are applied subsequently. Obtaining partial aggregations is usually called decomposition of aggregation functions, which is also an important technique in various fields related to aggregate query processing. In distributed group-by query processing, partial aggregation can be applied before the data shuffling phase [YGI09]. This is usually called initial reduce, with which the size of data transmission on a network can be substantially reduced. In multi-dimensional query processing, partial aggregation enables computing aggregation by merging summaries of cells with different granularity across multi-dimensional data, thereby enabling aggregate queries to be executed on precomputed results instead of base data [CD97]. An important point of query optimization in relational databases is to reduce intermediate result size for join [HMJJ00] and partial aggregations bring interests [YbL95] for group-by and join queries.

The current UDAF mechanism in data management or analytic system requires users to explicitly implement the routine of computing partial aggregation, the merging function and the finalizing function for an aggregation function. For example, to write a custom UDAF in Spark SQL [apad], 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 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 [GCoS+97]). Such a mechanism pushes from the side of a data management system to users the effort of obtaining partial aggregations from an aggregation function. Another issue underlying the current mechanism

of UDAFs is the loss of opportunities to optimize queries with UDAFs, since the semantics of a UDAF, i.e., what properties it has, may not be fully known by the query processor unless users explicitly define them. For example, PostgreSQL naturally supports parallel aggregation [Par19] for some built-in aggregates, i.e., SQL standard aggregates and statistical aggregates. However, it does not compute a UDAF in parallel, unless users explicitly tell the query processor that the UDAF is safe to be paralleled.

As explained previously, partial aggregations are the foundations of efficient computing an (algebraic) aggregation in a distributed architecture, but their computation results are seldom cached and reused to accelerate queries with aggregations, especially for UDAFs. It is well-known that evaluating queries from caches can be significantly faster than computing a query from base data. However, most previous works focus on *data dimension*, i.e., the predicate range of the second query is fully or partially contained in the first query, or the group-by granularity of the second query is bigger than the first query in OLAP applications. The *computation dimension*, i.e., sharing opportunities of different aggregation functions, are not widely taken into consideration, which can lead to the issue that caches can only be reused to queries which have the identical aggregation to the cached one. The general problem can be summarized as *how to compute an aggregate function from another one?* Built-in aggregations can be trivially handled since one can explicitly predefine their relationships of how to compute one from another one, e.g., avg can be computed from sum and count. However, when it comes to the scenarios of UDAFs, how to capture such a computation relationship is a hard nut to crack.

The general objectives of this thesis are twofold: firstly, we aim at automatically generating efficient partial aggregations from a declarative specification of UDAFs [ZTG17b, ZTG17a, ZTG18]. Secondly, we aim at identifying when partial aggregations of a UDAF can be reused to compute partial aggregations of another UDAF [ZTG18, ZT19]. The above objectives guided the design of SUDAF (Sharing User-Defined Aggregate Functions) [ZT19], a declarative framework that comes equipped with the ability to automatically generate parallel implementation from a mathematical expression of a UDAF and which supports efficient dynamic caching and reusing of partial aggregates. More precisely, SUDAF offers full flexibility to users by providing a declarative framework that allows them to write UDAFs as mathematical expressions and then use them in SQL statements. Then, SUDAF enables to decompose mathematical expressions of UDAFs and map them to efficient computations in massively parallel frameworks. Moreover, SUDAF dynamically caches partial aggregation results and checks whether partial aggregations in new UDAFs can be computed from the cached ones.

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

- We rely on the notion of a canonical form of UDAF [Coh06] to provide a generic implementation scheme for aggregate functions in massively parallel architectures. We identify a practical cost model of MapReduce computations, and we map the generic implementation schema of aggregations to such a cost model to have efficient MapReduce computations of aggregations.

- We build on the canonical form of aggregate functions to identify the right level of aggregation to keep in the cache. 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.

- We present SUDAF, a declarative UDAF framework that allows users to formulate a UDAF as a mathematical expression and use them in SQL statements. When executing a given query with UDAFs, SUDAF identifies appropriate partial aggregations from the mathematical expression of a UDAF and rewrites them using simple built-in functions of an underlying data management and analysis system.
- To deal with the undecidability of the *sharing problem*, we restrict the set of UDAFs supported by SUDAF. Three classes of predefined primitive functions are proposed: *primitive scalar functions*, *binary functions* and *primitive aggregate functions*. SUDAF also provides a composition operator that enables to create new functions by composing existing ones. This practical framework is powerful enough to be used in practical applications while it makes the sharing problem decidable. From a theoretical standpoint, we provide conditions to characterize the sharing problem in the SUDAF framework (Theorem 3). From a practical standpoint, we design an approach based on symbolic representations of mathematical expressions to efficiently verify the proposed conditions.
- We describe a sophisticated implementation of the SUDAF cache that allows maximizing the sharing possibilities while minimizing the redundancies in the cache. The SUDAF cache includes a symbolic index that captures the sharing relationships of symbolic classes of partial aggregations in UDAFs. The symbolic index is precomputed in an initialization step during the installation of SUDAF and then it is used to identify: (i) the unit of computations that are worth to cache, and (ii) when a cached result can be reused in the computation of a given UDAF.
- We implemented a SUDAF prototype, which can be used on top of existing data management and analysis systems. We report on experiments using SUDAF with both PostgreSQL and Spark SQL. Our experiments show that rewriting UDAFs using built-in aggregates can significantly speed up query execution time. Also, the proposed sharing technique can yield up to two orders of magnitude improvement in query execution time.

Thesis organization. The thesis is organized as follows. In Chapter 2, we introduce a canonical form of aggregation functions and discuss how it can be used to map UDAFs to massively parallel algorithms. In Chapter 3, we present the SUDAF practical framework. We first identify the right level of aggregation to keep in caches to enable efficient aggregate sharing across UDAFs. We formalize the problem of reusing partial computation results of aggregations as the sharing problem, and we show that it is an undecidable problem in general settings. Therefore, we propose primitive functions to restrict our study for practical aggregations. We propose in SUDAF for the sharing problem full categorization, complete sharing conditions and forms of reusing functions. In Chapter 4, we introduce a practical approach, based on symbolic representations of partial aggregates, to solve the sharing problem in the SUDAF framework. We also discuss the design principles underlying the caching scheme of SUDAF. At the end of the Chapter 4, we present an experimental evaluation of SUDAF on top of Spark SQL and PostgreSQL, and we respectively compare their performances for different sequences of aggregate queries. In Chapter 5, we first present the implementation details of SUDAF prototype, and then we discuss how to generate canonical forms from mathematical expressions of UDAFs. We discuss related works in Chapter 6 and conclude in Chapter 7.

Chapter 2

Parallelization of UDAFs

This chapter focuses on the following two problems related to the parallelization of UDAFs:

(i) *How to systematically map a UDAF to a distributed algorithm?* We identify a canonical form of UDAFs, well-formed aggregation [Coh06], which captures how a UDAF is constructed. The canonical form can be mapped to the MapReduce paradigm, which yields a generic MapReduce algorithm for computing UDAFs [ZTG17a].

(ii) *When the generated MapReduce algorithm is efficient?* We identify a class of efficient MapReduce algorithms, $\mathcal{MRC}$ algorithms [KSV10]. We analyze when the generic MapReduce algorithm for a UDAF obtained by its canonical form can be a $\mathcal{MRC}$ algorithm [ZTG17b, ZTG18].

2.1 A canonical form of UDAFs

In this section, we first present the canonical form of UDAFs used in this thesis, well-formed aggregation [Coh06]. We also show that an aggregate function having one of several algebraic properties can be systematically mapped to the canonical form.

An aggregate function takes as inputs several values and produces as an output a *single representative* value of the inputs [GMMP11]. In our work, we consider arbitrary aggregate functions operating on a multiset, $X = \{\{x_1, ..., x_n\}\}$. The size of X (the number of values in X) is denoted as $|X|$, $|X| = n, n \in \mathbb{N}^*$. Let D_s and D_t be two domains (i.e. a set of infinite number 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 [Coh06] to capture *the manner* in which a UDAF is created.

Definition 1. *(**Well-formed aggregation** [Coh06]) Let α be an aggregate function over the domain $\mathcal{M}(D_s)$ with the target domain D_t. We say that α is well-formed aggregation if there is a domain D_i and a triple $(F, \oplus, T)$ where*

- *$F : D_s \rightarrow D_i$ is a translating function;*
- *$\oplus$ is a commutative and associative binary operation over D_i and*
- *$T : D_i \rightarrow D_t$ is a terminating function;*

such that $\forall X \in \mathcal{M}(D_s), \alpha(X) = T(F(x_1) \oplus \ldots \oplus F(x_n))$ where $X = \{\{x_1, ..., x_n\}\}$.

F is a *scalar function*, i.e., a tuple at time function operating on values of some attributes of the same tuple. The binary operation $\oplus$ accumulates results of F and hence plays the role of an accumulator. T operates on the accumulated results of $\oplus$ to finalize

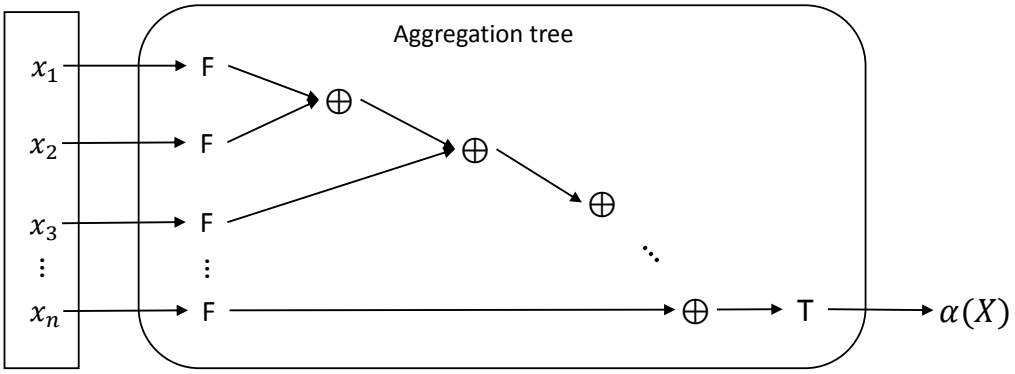

FIGURE 2.1: Data flow in well-formed aggregation

the computation of α. Figure 2.1 shows the data flow in the well-formed aggregation, which has a tree-like structure. The intermediate $\oplus$ nodes have the associative and commutative property, such that this tree-like structure can be any shape. In the sequel, we use $\sum_{i=1,\oplus}^{n} F(x_i)$ to denote $F(x_1) \oplus \ldots \oplus F(x_n)$, and when it is clear from the context we simply use $\sum F(x_i)$.

Example 1. *The aggregation average (avg),* $avg(X) = \frac{(\sum_{i=1}^{n} x_i)}{n}$, *can be expressed in the following canonical form, referred to as can-average form:*

- $F(x) = (x, 1)$;
- $(x, k) \oplus (x', k') = (x + x', k + k')$;
- $T(s_1, s_2) = s_1/s_2$.

In this thesis, we consider the well-formed aggregation as the canonical form of UDAFs. It should be noteworthy that a canonical form of an aggregate function α is not unique and any aggregation function has at least one canonical form [Coh06]. This latter one is given by the *trivial canonical form* $(F, \oplus, T)$, where F is the identity function, $\oplus$ is the set union and T is α. A trivial canonical form does not lead to efficient computation of aggregation since all corresponding data at worker nodes are sent to a unique master node where the aggregation is computed. While a non-trivial canonical form brings benefits since partial aggregation can be computed in parallel and only their computation results need to shuffle in a cluster.

We investigate below different algebraic properties of aggregation functions leading to a non-trivial canonical form.

- *Associative aggregation*. An aggregate function α is *associative* [GMMP11] if for any multiset $X = X_1 \cup X_2$, $\alpha(X) = \alpha(\alpha(X_1), \alpha(X_2))$. Associative and commutative aggregation function can be transformed into a canonical form $(F, \oplus, T)$ defined as follows where id denotes the identity function.

$$F = \alpha,\ \oplus = \alpha,\ T = id, \tag{2.1}$$

  For example, $\sqrt{\sum_{i=1}^{n} x_i^2}$ is an associative and commutative aggregate function, which has the following canonical form:

  - $F(x) = |x|$;
  - $|x| \oplus |x'| = \sqrt{x^2 + x'^2}$;

- $T(s) = s$.

- *Distributive aggregation*. An aggregation α is *distributive* [GCoS+97] if there exists a combining function C such that $\alpha(X) = C(\alpha(X_1), \alpha(X_2))$. Distributive and commutative aggregation can be mapped to the following canonical form:

$$F = \alpha,\ \oplus = C,\ T = id. \tag{2.2}$$

Commutative semi-group aggregation [CNS06] is another kind of aggregate function having the same behavior as commutative and distributive aggregation. An aggregation α is in this class if there exists a commutative semi-group $(H, \otimes)$, such that $\alpha(X) = \bigotimes_{x_i \in X} \alpha(x_i)$. The corresponding canonical aggregation $(F, \oplus, T)$ is illustrated as follows:

$$F = \alpha,\ \oplus = \otimes,\ T = id. \tag{2.3}$$

For example, $\sum_{i=1}^{n} (ln(x_i))^2$ is a distributive aggregate function, which has the following canonical form:

- $F(x) = (ln(x))^2$;
- $(ln(x))^2 \oplus (ln(x'))^2 = (ln(x))^2 + (ln(x'))^2$;
- $T(s) = s$.

- *Preassociative and commutative aggregation*. An aggregation α is *preassociative* [MLT15], if it satisfies $\alpha(Y) = \alpha(Y') \implies \alpha(X \cup Y \cup Z) = \alpha(X \cup Y' \cup Z)$. A kind of preassociative and commutative aggregation functions has the following form $\alpha(X) = \psi\left(\sum_{i=1}^{n} \varphi(x_i)\right)$, where ψ and φ are continuous and strictly monotonic unary functions (these functions satisfy the unarily quasi-range-idempotent and continuous property [MLT15]). A canonical form $(F, \oplus, T)$ for this kind of preassociative aggregation can be defined as following:

$$F = \varphi,\ \oplus = +,\ T = \psi. \tag{2.4}$$

For example, $\left(\sum_{i=1}^{n} x_i^3\right)^5$ is a preassociative and commutative aggregation, which has the following canonical form:

- $F(x) = x^3$;
- $x^3 \oplus x'^3 = x^3 + x'^3$;
- $T(s) = s^5$.

- *Quasi-arithmetic mean*. An aggregate function α is barycentrically associative [MLT16] if it satisfies $\alpha(X \cup Y \cup Z) = \alpha(X \cup Y' \cup Z)$, with $Y' = \{\underbrace{\{\alpha(Y), ..., \alpha(Y)\}}_{|Y|}\}$ where $|Y|$ denotes the number of elements contained in Y. A well-known class of commutative and barycentrically associative aggregation is quasi-arithmetic mean: $\alpha(X) = f^{-quasi}\left(\frac{\sum_{i=1}^{n} f(x_i)}{n}\right)$ where f is an unary function and f^{-quasi} is a quasi-inverse of f. With different choices of f, α can correspond to different kinds of mean functions, e.g., arithmetic mean, quadratic mean, harmonic mean etc. An immediate canonical form $(F, \oplus, T)$ of such functions is given by:

$$F = (f, 1),\ \oplus = (+, +),\ T = f^{-quasi}\left(\frac{\sum_{i=1}^{n} f(x_i)}{n}\right). \tag{2.5}$$

For example, $\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}$ is a quasi-arithmetic mean function, which has the following canonical form:

Aggregation	Formula	Canonical form $\left(\mathbf{F}, \oplus, \mathbf{T}\right)$
Count (n)	$\sum 1$	$\left((1), (+), s_1\right)$
Sum	$\sum x_i$	$\left((x_i), (+), s_1\right)$
Product	$\prod x_i$	$\left((x_i), (\times), s_1\right)$
Average (avg)	$\frac{\sum x_i}{n}$	$\left((x_i, 1), (+, +), \frac{s_1}{s_2}\right)$
Median	–	$\left((x_i), (\cup), median\right)$
Percentile	–	$\left((x_i), (\cup), percentile\right)$
Power mean	$(\frac{\sum (x_i)^p}{n})^{1/p}$	$\left((x_i^p, 1), (+, +), (\frac{s_1}{s_2})^{1/p}\right)$
Geometric mean	$(\prod x_i)^{1/n}$	$\left((x_i, 1), (\times, +), (s_1)^{1/s_2}\right)$
Variance	$\frac{\sum x_i^2}{n} - (\frac{\sum x_i}{n})^2$	$\left((x_i, x_i^2, 1), (+, +, +), \frac{s_2}{s_3} - (\frac{s_1}{s_3})^2\right)$
Stddev	$\sqrt{\frac{\sum x_i^2}{n} - (\frac{\sum x_i}{n})^2}$	$\left((x_i, x_i^2, 1), (+, +, +), \sqrt{\frac{s_2}{s_3} - (\frac{s_1}{s_3})^2}\right)$
Central moment	$\frac{\sum (x_i - avg)^k}{n}$	$\left(((x_i - avg)^k, 1), (+, +), s_1/s_2\right)$
LogSumExp	$ln(\sum exp(x_i))$	$\left((exp(x_i)), (+), ln(s_1)\right)$
Skewness	$\frac{(\sum (x_i - avg)^3)/n}{((\sum (x_i - avg)^2)/n)^{3/2}}$	$\left(((x_i - avg)^3, (x_i - avg)^2, 1), (+, +, +), \frac{s_1/s_3}{(s_2/s_3)^{3/2}}\right)$
Kurtosis	$\frac{(\sum (x_i - avg)^4)/n}{((\sum (x_i - avg)^2)/n)^2}$	$\left(((x_i - avg)^4, (x_i - avg)^2, 1), (+, +, +), \frac{s_1/s_3}{(s_2/s_3)^2}\right)$
Covariance	$\frac{\sum (x_i \times y_i)}{n} - \frac{\sum x_i \times \sum y_i}{n^2}$	$\left((x_i, y_i, x_i \times y_i, 1), (+, +, +, +), \frac{s_3}{s_4} - \frac{s_1 \times s_2}{s_4}\right)$
Correlation	$\frac{n \times \sum (x_i \times y_i) - \sum x_i \times \sum y_i}{\sqrt{n \times \sum x_i^2 - (\sum x_i)^2} \times \sqrt{n \times \sum y_i^2 - (\sum y_i)^2}}$	$\left((x_i, x_i^2, y_i, y_i^2, x_i \times y_i, 1), (+, +, +, +, +, +), \frac{s_6 \times s_5 - s_1 \times s_3}{\sqrt{s_6 \times s_2 - (s_1)^2} \times \sqrt{s_6 \times s_4 - (s_4)^2}}\right)$
ϵ **quantile** [GDT+18]	$\text{MS}\left(min, max, n, \sum x_i, ..., \sum x_i^{k_1}, \sum ln(x_i), ... \sum ln^{k_2}(x_i)\right)$	$\left((x_i, x_i, 1, x_i, ..., x_i^{k_1}, ln(x_i), ..., ln^{k_2}(x_i)), (min_2, max_2, +, ..., +), \text{MS}\,(s_1, ..., s_{3+k_1+k_2})\right)$

TABLE 2.1: Examples of aggregation in canonical forms.

- $F(x) = x^2$;
- $x^2 \oplus x'^2 = x^2 + x'^2$;
- $T(s) = \sqrt{s^2}$.

We also list some practical aggregate functions with canonical forms in Table 2.1, where outputs of $\oplus$ (the input of T) are denoted as a sequence $(s_1, ..., s_m)$. It is interesting to be noteworthy that practical aggregates usually have addition or product as an element of $\oplus$ function in their non-trivial canonical forms. While holistic aggregation functions [GCoS+97] can only have trivial canonical forms (median and percentiles).

In the rest of this work, we will explore canonical forms to generate an efficient module for computing UDAFs in a distributed and parallel architecture (c.f., Section 2.2). We will also design a share mechanism for accelerating the computation of various UDAFs (c.f., Chapter 3). We are also aware that it is not realistic to ask a user to provide UDAFs in their canonical forms. Since a mathematical expression is a natural way to declare a UDAF, we allow users to formulate UDAFs as mathematical expressions. We design an approach to automatically generate a canonical form of a UDAF from its mathematical expression (c.f., Section 5.3). As a side effect, the semantics of UDAFs can be captured and exploited to share computations of UDAFs.

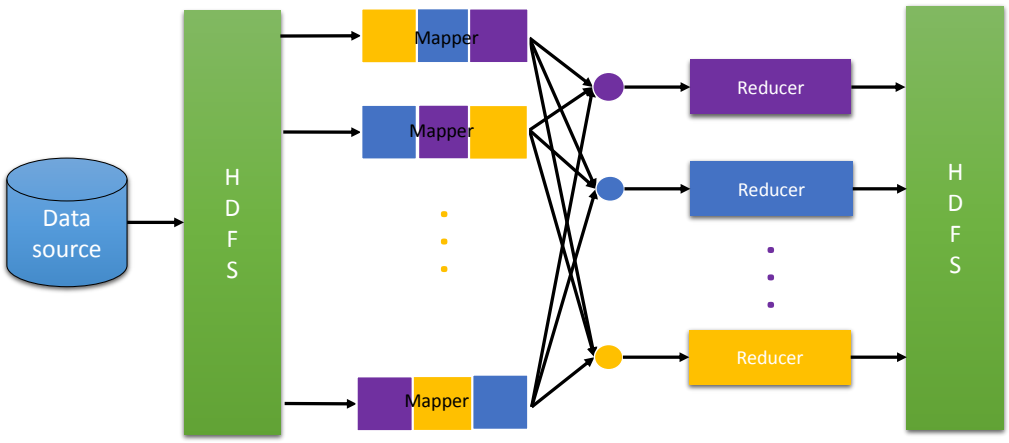

FIGURE 2.2: Data flow in the MapReduce framework.

2.2 Mapping a canonical form to a massively parallel algorithm

This section describes how a UDAF given in its canonical form can be mapped into a MapReduce algorithm and discusses the efficiency of the algorithm. More precisely, we first recall the MapReduce framework, then we investigate associated cost models and identify one of them. Finally, we present a generic MapReduce algorithm based on the canonical form and study its efficiency according to the identified cost model.

2.2.1 The MapReduce computation framework

The MapReduce [DG04] framework is designed for massively parallel computing over a cluster of commodity machines. A MapReduce program consists of one or several rounds of computations, and every round is made up of three phases, the mapper phase, shuffle phase and reducer phase. At the mapper phase, a tuple-at-a-time function is applied on every line of input data, and a binary function is applied to aggregate mapper outputs at the reducer phase. The shuffle phase is proceeded between the mapper and reducer phase. Given an input of key-value pairs $\{(k, v)...\}$, mapper outputs having identical keys at different machines need to be shuffled to one machine to complete the reducer phase. Data shuffling is usually a bottleneck for improving the performance of MapReduce algorithm. We present the data flow in the MapReduce framework in Figure 2.2.

2.2.2 Cost models of massively parallel algorithms

We investigate widely used cost models of MapReduce or related computing frameworks, and we choose one of them to ensure the efficiency of our approach. In order to illustrate the differences among them, we make the following denotations to show their different assumptions on parameters in distributed computing. Assuming given a cluster of P machines, let $\mathcal{A}$ be the MapReduce program executed on this cluster in rounds (one round or multiple rounds). The input of $\mathcal{A}$ is a finite sequence of pairs $< k_j; v_j >$, for $j \in [1, n]$ where k_j and v_j are binary strings. Hence, $\mathcal{A}$ operates on the input of length $nb = \sum_{j=1}^{n} (|k_j| + |v_j|)$ bits. During each round, let C and M be separately the computation time and machine space at each node.

The MUD algorithm. The massive, unordered, distributed (mud) algorithmic model [FMS+10] transforms symmetric sequential algorithm to parallel algorithm. It fixes communication and space to be $polylog(nb)$, and its corresponding time complexity is $\Omega(2^{polylog(nb)})$. We summarize MUD's constraints in the following,

- $C = \Omega(2^{polylog(nb)})$;
- $M = polylog(nb)$.

Memory Bound MapReduce algorithm. The trade-off between round complexity and reducer space complexity is analyzed in [GSZ11]. In order to exploit the benefits of parallelism and have decent round complexity, it is necessary to bound reducer space. Unlike the other works, they did not restrict reducer memory to a fixed bound at the beginning. However, after analyzing several complex problems e.g., prefix sums and multi-searching, they show that a reducer memory of $\Theta(n^{1-\epsilon})$ for some constant $\epsilon > 0$ can solve these problems with high probability. We summarize this as follows,

- $C = null$;
- $M = \Theta(n^{1-\epsilon})$, $\epsilon > 0$.

The MPC model. In [BKS17], the massively parallel communication model was proposed to analyze the trade-off between communication load and computation rounds. Specifically, MPC was used to analyze the distributed processing of conjunctive queries in two situations, the communication load required to process a given query in one round and the computation rounds required to process a given query with fixed maximum load. The constraints made by MPC can be summarized as follows,

- $C = null$;
- $M = \mathcal{O}((nb)/p^{1-\epsilon})$, $\epsilon \in [0, 1]$.

The $\mathcal{MRC}$ model. In [KSV10], a model of efficient computation using MapReduce paradigm is introduced. The proposed model limits the number of machines and the space per machine to be sub-linear in the length of the input. More precisely, [KSV10] defines the MapReduce class, noted $\mathcal{MRC}$, as the class of efficient MapReduce programs. Then $\mathcal{A}$ belongs to the class $\mathcal{MRC}^i$, if $\mathcal{A}$ satisfies the following constraints:

- $M = \mathcal{O}((nb)^{1-\epsilon})$, $\epsilon > 0$;
- $C = \mathcal{O}((nb)^k)$, for some constant k;
- $P = \mathcal{O}((nb)^{1-\epsilon})$, $\epsilon > 0$;
- $R = \mathcal{O}(log^i(nb))$.

Our goal is to identify when an aggregation function can be efficiently processed using their canonical forms in one round, therefore our choice focuses on a cost model of efficient MapReduce algorithm with realistic assumptions. The *MUD* [FMS+10] model does not restrict on computation time, which disqualifies *MUD* as a possible candidate in our context. The model in [GSZ11] relaxes reducer memory to be an arbitrary number to solve complex problems, e.g., prefix sums and multi-searching. The *MPC* model mainly focuses on analyzing the trade-off between communication load and computation rounds for conjunctive queries. Finally, the *MRC* model considers necessary parameters

MapReduce phase	Operation
mapper	$\sum_{\oplus} F(x_i)$
combiner	$\oplus$
reducer	$T(\sum_{\oplus} O_i)$

TABLE 2.2: $MR(\alpha)$: a generic MR algorithm for UDAFs.

for parallel computing, e.g., communication time, machine space and computing time, and it also makes more realistic assumptions to have a generic algorithm to compute UDAFs. Hence, a MapReduce algorithm satisfying the $\mathcal{MRC}$ constraints is considered as an efficient parallel algorithm and will be called hereafter a $\mathcal{MRC}$ algorithm.

2.2.3 $MR(\alpha)$: a MapReduce algorithm to compute UDAFs

The canonical form provides a generic plan for processing aggregation in massively parallel architectures. Indeed, given an aggregate function $\alpha = (F, \oplus, T)$, 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 aggregated using $\oplus$ and T to produce the final result $\alpha(X)$. For example, a MapReduce-based implementation of α can be straightforwardly derived from its canonical form $\alpha = (F, \oplus, T)$: processing F and $\oplus$ at mapper, $\oplus$ at combiner, and $\oplus$ and T at reducer. Table 2.2 depicts the corresponding generic MapReduce algorithm, $MR(\alpha)$, to compute $\alpha(X)$, where a mapper input is a multiset $X_i \subseteq X$, and the output of a mapper i is denoted by O_i.

According to the above discussion, every aggregation function α given in a canonical form $(F, \oplus, T)$ can be turned into a MapReduce algorithm $MR(\alpha)$. However, the generated $MR(\alpha)$ algorithm is not necessarily an efficient algorithm (i.e., a $\mathcal{MRC}$ algorithm). For example, the algorithm MR(can-average), derived from the can-average form of the *average* function is a $\mathcal{MRC}$ algorithm. While, the algorithm $MR(naiveaverage)$, derived from the trivial canonical form of *average*, is not a $\mathcal{MRC}$ algorithm. Indeed, in the naive canonical form of *average*, F is the identity function and $\oplus$ is the multiset union. Hence, the total size of the output of mappers is equal to nb (the length of the input), and if, in a worst case, all the mapper outputs are sent to only one reducer, the reducer will need a space equal to nb. However, a $\mathcal{MRC}$ algorithm requires every reducer uses a sub-linear space in nb. Therefore, we address in the sequel the following question:

Given an aggregation function α in its canonical form, when the corresponding $MR(\alpha)$ is efficient?

In other words, we are interested in characterizing under which conditions it can be ensured that $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

Before answering to the above question, we first observe that a $MR(\alpha)$ for a practical aggregate function α (i.e., examples in Table 2.1) has the following two properties:

(1) a $MR(\alpha)$ is a one-round MR algorithm, which makes a $MR(\alpha)$ always satisfy the condition of computation round in $\mathcal{MRC}$ model;

(2) mapper or reducer computations of a practical $MR(\alpha)$ take no longer time than $O((nb)^k)$, which is the upper bound of local computation time in $\mathcal{MRC}$ model.

For example, the algorithm $MR(naiveaverage)$, derived from the trivial canonical form of *average*, indeed has these two properties, and we know that $MR(naiveaverage)$ is not a $\mathcal{MRC}$ algorithm because the length of its intermediate results is too large to be stored in a machine with space M when all the mapper outputs are sent to a unique reducer.

In order to identify when a $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm, we consider the following settings for a $MR(\alpha)$:

(1) A $MR(\alpha)$ is a one-round MR algorithm.

(2) The mapper and reducer program of a $MR(\alpha)$ operate in $O((nb)^k)$ time.

(3) We consider all mapper outputs are sent to one reducer in the computation of $MR(\alpha)$, which can be seen as the extremely worst (EW) situation in a MapReduce computation. It is trivial to see that if $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm under the EW case, then $MR(\alpha)$ will be a $\mathcal{MRC}$ algorithm in general cases.

(4) We consider a $MR(\alpha)$ with an input X is computed in a cluster of P machines containing M space for each machine, which is referred to as $\mathcal{MRC}$ *environment* $E = (X, P, M)$, where $X = \{<k_j; v_j>, \text{ for } j \in [1, n]\}$ is an input of $nb = \sum_{j=1}^{n}(|k_j| + |v_j|)$ bits ($|v_j|$ is the length of v_i, similar for $|k_j|$), the number of machines P in a MapReduce cluster is in $O((nb)^{1-\epsilon_1}), \epsilon_1 > 0$, and the storage space for each machine in the cluster is in $O((nb)^{1-\epsilon_2}), \epsilon_2 > 0$.

Note that, in the $\mathcal{MRC}$ *environment*, both the upper bound of P and M are indeed the upper bound of machine number and machine space in $\mathcal{MRC}$ model. Then, the original problem, whether a $MR(\alpha)$ for a practical aggregate function α is a $\mathcal{MRC}$ algorithm, can be reduced to *whether the* $MR(\alpha)$ *can be successfully computed in* $\mathcal{MRC}$ *environment* $E = (X, P, M)$ *under the EW situation*.

We propose the following lemma to identify when a $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm. More precisely, the following lemma states that the output length of every mapper needs to be bounded by a space that is sub-linear in nb. This ensures that the length of the mapper outputs is smaller enough to make the underlying computation possible at one machine during the reducer phase under the EW case.

Lemma 1. *Let* $MR(\alpha)$ *be a MR algorithm of an aggregation function* α *and* $E = (X, P, M)$ *be a* $\mathcal{MRC}$ *environment. Then,* $MR(\alpha)$ *is a* $\mathcal{MRC}$ *algorithm, if the output length of the mapper function in* $MR(\alpha)$ *is in* $O((nb)^{1-\zeta})$ *with* $\zeta \geqslant max(\epsilon_2, 1 - \epsilon_1 + \epsilon_2)$.

Proof. W.l.o.g, we consider every machine in the $\mathcal{MRC}$ environment will execute one mapper. Then, the upper bound of mapper output length, $O((nb)^{1-\zeta})$ with $\zeta \geqslant max(\epsilon_2, 1 - \epsilon_1 + \epsilon_2)$, ensures that $MR(\alpha)$ can be successfully computed in a $\mathcal{MRC}$ environment under the EW situation. Indeed, $O((nb)^{1-\zeta}) \leqslant O((nb)^{1-\epsilon_2})$, because $\zeta \geqslant \epsilon_2$. Then a mapper output does not overflow a mapper space. Moreover, for the length of total mapper outputs $P \times O((nb)^{1-\zeta})$, we have $P \times O((nb)^{1-\zeta}) \leqslant M$, because $\zeta \geqslant 1 - \epsilon_1 + \epsilon_2$. Then, $MR(\alpha)$ can be computed in the $\mathcal{MRC}$ environment under the EW case (all mapper outputs are sent to one reducer). Therefore, according to the $\mathcal{MRC}$ constraints, we have $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm. □

Until now, we have seen when $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm, that the output length of $\sum_{\oplus} F(x_i)$ needs to be bounded by $O((nb)^{1-\zeta})$ with $\zeta \geqslant max(\epsilon_2, 1 - \epsilon_1 + \epsilon_2)$. In the follows, we shall discuss which practical aggregate functions can satisfy this condition. We observe that $\oplus$ function plays a more important role than F function in the above condition, i.e., in making $\sum_{\oplus} F(x_i)$ satisfy the condition in Lemma 1. Indeed, in the computation of $\sum_{\oplus} F(x_i)$, F is applied on every individual value, i.e., a scalar function, and $\oplus$ is applied to accumulate values. Hence, the output length of $\sum_{\oplus} F(x_i)$ is mainly determined by applying the $\oplus$ function, i.e., the length increasing by using $\oplus$ to accumulate values.

According to the canonical forms of most common used aggregate functions (see Table 2.1), the $\oplus$ function in canonical forms can be one of multiset union, addition, and multiplication, or a tuple of their combinations, e.g., $\oplus = (+, +)$ for *can-average*. W.l.o.g,

we consider the case that $\oplus$ is multiset union, addition, or multiplication since the other cases can be trivially reduced to this case.

It is trivial to see that a $MR(\alpha)$ for an aggregate function having set union as $\oplus$ is not efficient because of shuffling all input values from mappers to a reducer. In the following, we study the left two cases, i.e. when $\oplus = +$ or $\oplus = \times$.

It is noteworthy that, in practice, there is always a trade-off between the length of results and computation precision. For example, in Java, the primitive data type double has a fixed 64-bit length, but in this case, the precision is out of control. While, using Big Decimal one can obtain arbitrary precision in arithmetic computations, but the computation result has an unfixed length. For the case of using fixed-length data type, a $MR(\alpha)$ with $\oplus = +$ or $\oplus = \times$ is always a $\mathcal{MRC}$ algorithm, because there is no increase in the length of intermediate results when accumulating values, in other words, the length of a mapper output is always the fixed length of data type. Our analysis will focus on computing a $MR(\alpha)$ using unfixed data type, which are the cases of computing with arbitrary precision, or exact computing with unlimited precision.

$MR(\alpha)$ with $\oplus = +$. We identify when a $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm for the case that $\oplus = +$ in the following theorem.

Theorem 1. *Let $MR(\alpha)$ be a MR algorithm of an aggregation function α given in a canonical form with $\oplus = +$ and $E = (X, P, M)$ be a $\mathcal{MRC}$ environment. Then, $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm, if the length of $F(v_{max})$, where v_{max} is the maximum value in X, is bounded by $O((nb)^{1-\zeta})$ with $\zeta \geqslant max(\epsilon_2,\ 1 - \epsilon_1 + \epsilon_2)$.*

Proof. Let v_{max} be the maximum value in X. Assuming mapper i receives l_i values, and the worst case of accumulating these l_i values with $+$ is that every value v_i is the maximum value v_{max}, and the corresponding result is $l_i \times F(v_{max})$. Then, the maximum length of a mapper output is $|O_i| = log(F(v_{max}) \times l_i) = log(F(v_{max})) + log(l_i)$. In fact, we have $log(l_i) = log(\frac{M}{log(F(v_{max}))}) < log(M) = log(O((nb)^{1-\epsilon_2}))$. Because $log(F(v_{max}))$ is bounded by $O((nb)^{1-\zeta})$ with $\zeta \geqslant max(\epsilon_2,\ 1 - \epsilon_1 + \epsilon_2)$, such that $|O_i|$ is bounded by $log(O((nb)^{1-\epsilon_2})) + O((nb)^{1-\zeta}) = O((nb)^{1-\zeta})$. According to Lemma 1, $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm. □

The upper bound of the length of $F(v_{max})$ in Theorem 1 is quite generic, which can be $\frac{M}{P}$ with respect to a $\mathcal{MRC}$ environment. In practice, the machine space in bits divided by the number of machines in a cluster can be quite large. Then, we can conclude: $MR(\alpha)$ with $\oplus = +$ is a $\mathcal{MRC}$ algorithm under most realistic cases.

$MR(\alpha)$ with $\oplus = \times$. There is a general belief that irrespective of the underlying computation architecture, if an aggregation function is associative and commutative, then the corresponding partial aggregation can be efficiently processed. However, this widespread practice is not correct with the consideration of the $\mathcal{MRC}$ cost model. Let us consider the computation of an aggregation function $\alpha(X) = \prod v_i$, which is indeed associative and commutative, and let input $X = \{v_i, \forall i \in [1, n]\}$ where v_i is a binary string. W.l.o.g., assume $v_i > 0$. The total length of the input for α is $\sum log(v_i), \forall v_i \in X$. α is indeed commutative and associative. Hence the partial aggregation $\alpha(X_i)$, where $X_i \in X$ containing l_i values, can be computed at the **Accumulator**. We ignore the **Combiner** phase since it does not impact our reasoning. The computation results of a partial aggregation $\alpha(X_i)$ is $\prod v_j, \forall v_j \in X_i$, of which encoding requires $log(\prod v_j) = \sum log(v_j), \forall v_j \in X_i$, bits. In the worst case of MapReduce, all mapper results (the case of one mapper at each machine)

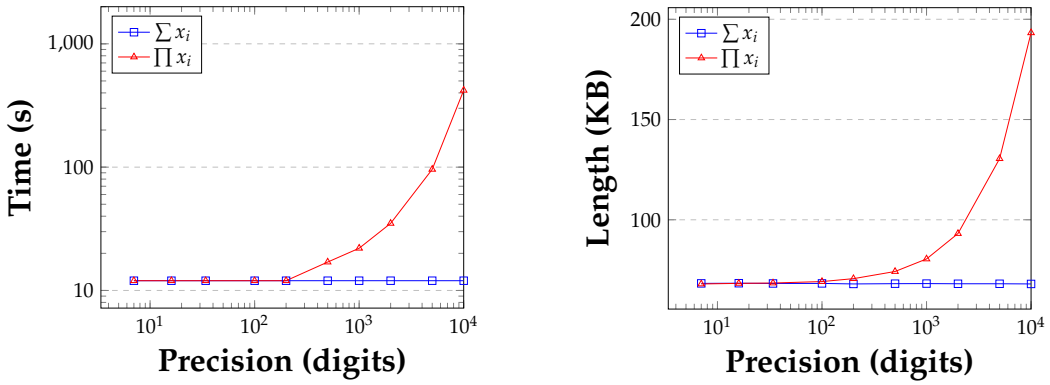

FIGURE 2.3: Computation time and total partial result length of $\sum x_i$ and $\prod x_i$ with different significant digits.

	$\sum x_i$	$\prod x_i$
Time (s)	12	1560
Size (KB)	$6.8097e+4$	$3.6793029e+7$

TABLE 2.3: Computation time and total partial result size of $\sum x_i$ and $\prod x_i$ with **unlimited precision**.

are sent to a reducer. Hence the reducer will need a space equal to $\sum log(v_i), \forall v_i \in X$. In another sense, the computation of the product contains shuffling all input data on all mappers to one reducer, which is not a $\mathcal{MRC}$ algorithm.

As a conclusion of this section, in the case where a $MR(\alpha)$ is computed using unfixed data types, we can have the following results:

- If the $\oplus$ function of α contains a multiset union operator $\cup$ or an arithmetic multiplication $\times$, then a corresponding $MR(\alpha)$ is not a $\mathcal{MRC}$ algorithm.
- If the $\oplus$ function of α is a tuple of arithmetic additions $+$, then a corresponding $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm in most realistic situations, i.e., the computation of $MR(\alpha)$ satisfies the condition in Theorem 1.

2.3 Experimental evaluation.

Based on the above analysis, we compare computation time and partial result length in computing $\sum x_i$ and $\prod x_i$ using an arbitrary precision. We use the store_sales table from TPC-DS generated by scale 10. We program the following two queries in the way of Spark RDD with a different predefined precision and unlimited precision.

```
SELECT Sum(ss_sales_price)  FROM store_sales WHERE ss_sales_price != 0;
SELECT Prod(ss_sales_price) FROM store_sales WHERE ss_sales_price != 0;
```

We run the above two queries on a spark cluster containing one master node and six slave nodes running Spark 2.2.0. The experiment results for the case of predefined precision is shown in Figure 2.3. One can observe that the computation time and the length of all mapper results of computing $\prod x_i$ are exponentially increasing with respect to the number of digits used to store final computation results. However, in the computation of $\sum x_i$, the computation time and the length of all mapper results do not increase with the number of digits. Table 2.3 shows the computation time and partial result length in the

case of unlimited precision (exact computing). One can observe that computation time of $\prod x_i$ is two orders of magnitudes larger than the time of $\sum x_i$, and all the partial result length of $\prod x_i$ is three orders of magnitudes larger than that of $\sum x_i$.

To be summarized, both $\sum x_i$ and $\prod x_i$ can be efficiently processed using fewer digits in precision. When using more digits, or even unlimited digit, $\sum x_i$ stays on the same execution time and result length, while $\prod x_i$ will dramatically increase in terms of computation time and result length.

2.4 Summary

- We identify the well-formed aggregation function, which provides a generic form of user-defined aggregation functions. We show that some well-known algebraic aggregation functions can be systematically mapped into the well-formed aggregation.
- We consider the well-formed aggregation function as a canonical form of UDAFs, based on which we present a generic Map-Reduce algorithm $MR(\alpha)$ to compute UDAFs.
- We identify the upper bound of the length of $\sum_{\oplus} F(x_i)$ for a UDAF α given in a canonical form $(F, \oplus, T)$, such that the corresponding $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm (a class of efficient MapReduce algorithm). We further study whether a practical aggregate function given in a canonical form $(F, \oplus, T)$, where $\oplus$ can only be one of $\{\cup, \times, +\}$, is a $\mathcal{MRC}$ algorithm. We show that when $\oplus$ is one of $\{\cup, \times\}$, $MR(\alpha)$ cannot be a MR algorithm. While, if $\oplus = +$, it is trivial for $MR(\alpha)$ to be a $\mathcal{MRC}$ algorithm.

Chapter 3

The sharing problem in SUDAF framework

In this chapter, we focus on speeding up the execution of queries with UDAFs by reusing cached answers to previous queries during the evaluation of new queries. Specifically, we deal with the following issues:

(i) *What data should be cached in order to optimize the evaluation of UDAF?* We analyze all possible caching choices based on the pipelines provided in canonical forms of UDAFs, and we choose to cache partial aggregation results [ZTG18, ZT19].

(ii) *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 [ZTG18, ZT19].

(iii) *How to deal with the undecidability of the sharing problem for practical aggregations?* We solve the problem in the context of practical aggregate functions. We propose SUDAF (sharing user-defined aggregate functions), which is an expression-based aggregation framework, i.e., creating a UDAF by declaring a mathematical expression of a UDAF [ZT19].

SUDAF relies on the canonical form of aggregations that enable efficient execution and sharing of UDAFs. The proposed practical framework is powerful enough to be useful in many real-world applications while it makes the sharing problem decidable [ZT19].

3.1 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 various UDAFs. In the following example, we consider 4 relations of the TPC-DS [NP06] 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 $\theta_0(X,Y) = avg(Y) - \theta_1 avg(X)$.

To be able to use θ_1 in an SQL statement, one needs to hard-code it as a user-defined function, e.g., in Spark SQL, one writes a piece of Java or Scala code to create θ_1. Assume that a hard-coded user-defined function `theta1()`, that implements the function $\theta_1()$, is created and the following query Q1 is issued (the corresponding query plan is shown in Figure 3.3):

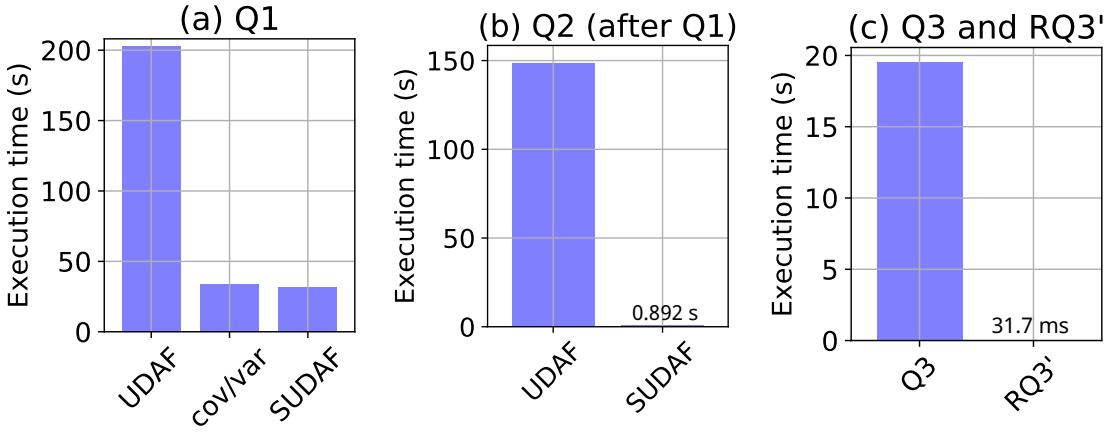

FIGURE 3.1: Experiments in **PostgreSQL** with the TPC-DS dataset (scale = 20). UDAFs theta1() and qm() are created in PL/pgSQL.

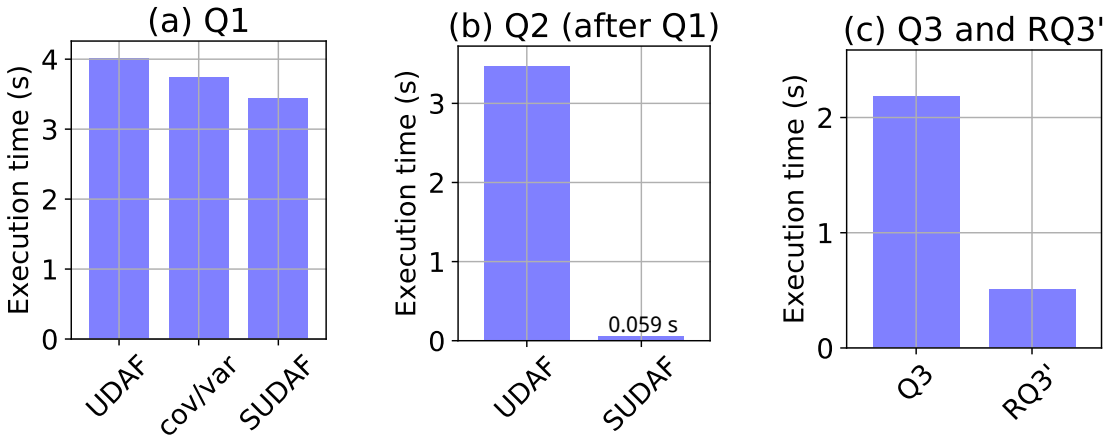

FIGURE 3.2: Experiments in **Spark SQL** with the TPC-DS dataset (scale = 100). UDAFs theta1() and qm() are created using UserDefinedAggregateFunction in Scala.

```
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 needs of any programming effort. This way of defining theta1() is much easier and more compact compared to the procedural way of hard-coding a user-defined function, for example, using Java or Scala programming languages. The aggregation function average can also be created in a similar way, i.e., $avg(X) = \sum x_i / \sum 1$.

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 UDAFs theta1() and avg() used in the query Q1 are rewritten into a set of partial aggregates using the built-in function *sum* and *count*, and (ii) the partial aggregates computed during the execution of Q1 can be cached and reused to compute various other UDAFs.

SELECT item_sk, year, avg(ss_list_price),
avg(ss_sales_price),
theta1(list_price, sales_price)
GROUP BY item_sk, year

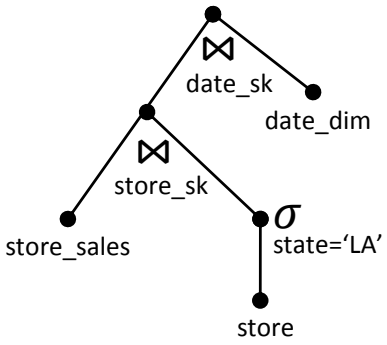

FIGURE 3.3: Plan of the query Q1.

SELECT ss_item_sk, d_year, count(*) s1, sum(list_price) s2,
sum(pow(list_price),2) s3, sum(sales_price) s4,
sum(list_price * sales_price) s5
GROUP BY item_sk,year

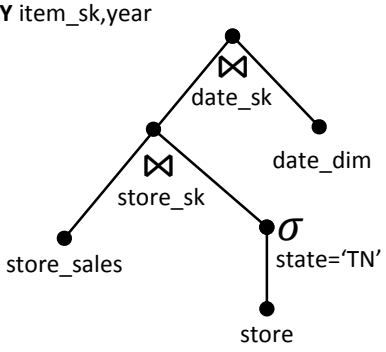

FIGURE 3.4: Plan of the subquery of RQ1.

Rewriting UDAFs 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. 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 (the corresponding query plan is shown in Figure 3.8) where the partial aggregates are first computed and then theta1() is computed using the partial aggregates as follows: $\theta_1 = \frac{s_1 s_5 - s_4 s_2}{s_1 s_3 - (s_2)^2}$.

```
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 hard-coded user-defined functions. Figure 3.1 (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 3.2 (a) using SUDAF on top of Spark SQL, where Q1 is $0.8X$ 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 more 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 3.1 (a) and Figure 3.2 (a), which is at a same order of magnitude as SUDAF execution time. However,

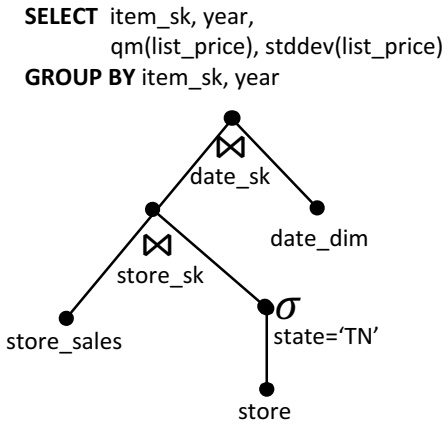

FIGURE 3.5: Plan of the query Q2.

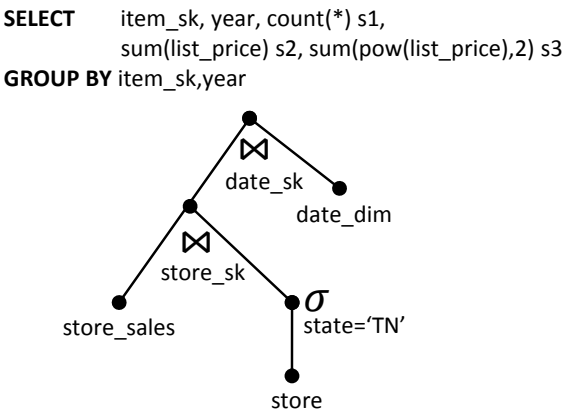

FIGURE 3.6: Plan of the subquery of RQ2.

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 below, the partial aggregates computed by SUDAF open wider sharing possibilities than the variance and the covariance functions.

Sharing partial aggregations between UDAFs. Caching the result of Q1, which contains the aggregate values of theta1(), is of little interest from the computation sharing perspective. However, the partial aggregates $s_1, \ldots, s_5$ computed by the query RQ1 offer more possibilities to be reused in future UDAFs computations. We illustrate the sharing idea by the following example. Consider a new query Q2 (the corresponding plan is shown in Figure 3.5) 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() and stddev() are defined using the mathematical expressions given in Table 2.1. 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;
```

Consequently, if the partial aggregates of RQ1 are kept in a cache or materialized, RQ2 can reuse the results of these partial aggregates to avoid computing from base data. This makes the execution of RQ2 significantly faster than the execution of the original

query Q2. We report the query time of Q2 when it is executed by SUDAF on top of PostgreSQL in Figure 3.1 (b) and by SUDAF on top of Spark SQL in Figure 3.2 (b). In both figures, the execution time of SUDAF is compared w.r.t. the execution time of the query Q2, 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 possibilities between these two queries.

It is also noteworthy 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 [DRSN98, WWDI17] 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 into queries over the data chunks. Extending SUDAF with such techniques enables to share partial aggregates over predefined data chunks.

However, the query processing techniques using data chunks can be performed only on restricted classes of queries, e.g., simple range queries without join in [WWDI17] and OLAP queries, which are queries expressed over the predefined dimensions of a multidimensional model, in [DRSN98]. As explained below, to handle more expressive queries, we envision the use of SUDAF to extend existing query rewriting using aggregated views algorithms (e.g., see [CNS99, CNS00]).

Extending query rewriting using aggregate views. We show that factoring out partial aggregations of UDAFs can improve traditional query rewriting using aggregate views algorithms. Assume that 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
```

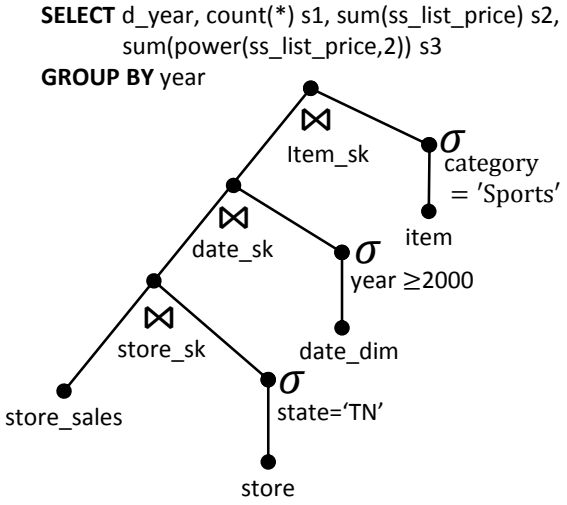

FIGURE 3.7: Plan of the subquery of RQ3.

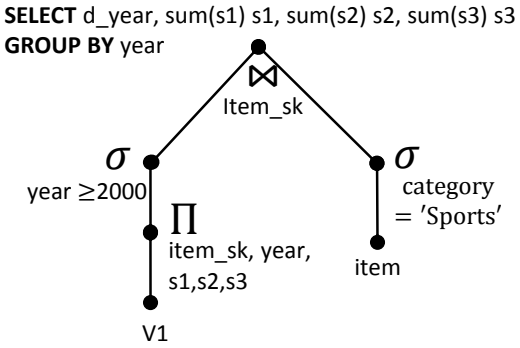

FIGURE 3.8: Plan of the subquery of RQ3'.

```
                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 [CNS00] to rewrite the subquery of RQ3 using V1. The obtained rewriting, denoted by RQ3', is shown below.

```
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 [CNS00] supports only the *sum* and *count* aggregates). We report the execution time of Q3 and RQ3' in PostgreSQL in Figure 3.1 (c) and Spark SQL in Figure 3.2 (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 chapter, we address the following challenges:

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

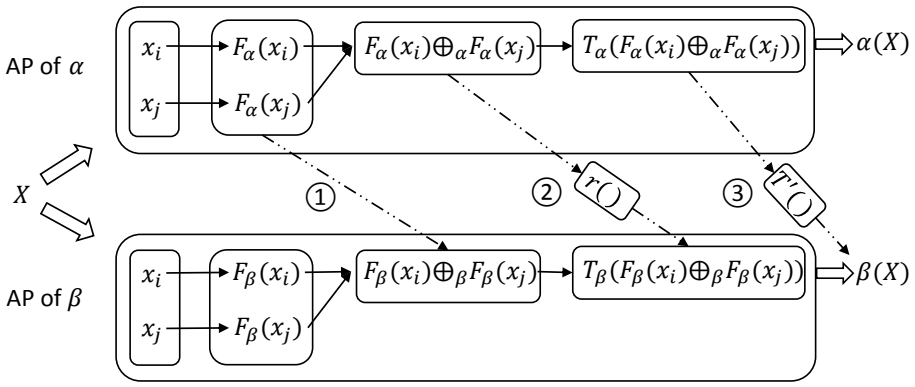

FIGURE 3.9: Sharing aggregation pipeline (AP).

3.2 Caching and sharing aggregation

We rely on a canonical form $(F, \oplus, T)$ to analyze which data should be cached such to obtain more possibilities to reuse caching results, then we formalize the sharing problem.

3.2.1 Caching aggregate data

As we explained above, the first issue is related to the identification of the right level of aggregation to keep in the cache. We build on the canonical form of aggregate functions to address this issue. As illustrated below, the canonical form provides a *natural* way to identify the adequate aggregation level that is worth to cache. As an example, Figure 3.9 depicts the workflows corresponding to the executions of two aggregate functions $\alpha = (F_\alpha, \oplus_\alpha, T_\alpha)$ and $\beta = (F_\beta, \oplus_\beta, T_\beta)$. Consider a scenario where an implementation of α based on its canonical form is executed first. Then, as shown in Figure 3.9, when the UDAF β is evaluated, there are three possibilities to reuse (partial) computations of α, respectively, by caching the results of F_α, the result of $\sum_{\oplus_\alpha} F_\alpha$, or the one of α. It is clear that storing the result of F_α (flow (1) in Figure 3.9) does not provide any added value to the computation of β since F_α is a scalar function. Similarly, storing the final result of α (flow (3) in Figure 3.9), computed by T_α, is of little interest as it offers very restricted possibilities of re-use of the cached result in the computation of other UDAFs. In fact, T_α should not be expected to have an inverse function [Coh06] since generally, it has multiple variables as inputs. However, the partial aggregation $\sum_{\oplus_\alpha} F_\alpha$ (flow (2) in Figure 3.9) offers much more reuse potentials than the third one. For example, if α is *variance* and β is *power mean* $(p = 2)$, hence it is not easy to reuse the final result of α to compute β. However, using the canonical forms described at Table 2.1, one can observe that it is rather straightforward to efficiently compute the partial aggregation of β from a partial aggregation of α. Therefore, we compute and cache $\sum_{\oplus_\alpha} F_\alpha(x_i)$, and the general optimizing problem is then to explore the possibility of the existence of a function r, such that $\beta(X)$ can be computed from $\sum_{\oplus_\alpha} F_\alpha(x_i)$, i.e., $\beta(X) = T_\beta(r(\sum_{\oplus_\alpha} F_\alpha(x_i)))$.

3.2.2 Sharing aggregation states

Let α be an aggregate function given in a canonical form $(F, \oplus, T)$ and $\sum_\oplus F(x_i)$ be the partial aggregation of α. As shown in Table 2.1, a typical $\oplus$ operator produces as an output a tuple of (partially) aggregated results. Hence, a partial aggregation could be written as $\sum_\oplus F(x_i) = \left(\sum_{\oplus_1} f_1(x_i), ..., \sum_{\oplus_m} f_m(x_i)\right)$, where the f_is are scalar functions and the $\oplus_i$s are commutative and associative binary operations. For example, in the *canaverage* form, the binary operation $\oplus = (+, +)$ produces as an output a pair of aggregate values where

the first component sums the values and the second component counts the number of elements.

In the sequel, for an aggregate function $\alpha = (F, \oplus, T)$ over a multiset X, with the following form of partial aggregation:

$$\sum_{\oplus} F(x_i) = \big(\sum_{\oplus_1} f_1(x_i), ..., \sum_{\oplus_m} f_m(x_i)\big),$$

we denote the component $s_j(X) = \sum_{\oplus_j} f_j(x_i)$ a (partial) *aggregation state* of α.

We rely on the notion of aggregation state 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 2. *Let $s'(X)$ and $s(X)$ be respectively two aggregation states of two UDAFs. Then, s' shares s if and only if there exists a computable function r such that*

$$s'(X) = r \circ s(X), \forall X \in \mathcal{M}(D). \tag{3.1}$$

Note that, if $s'(X) = s(X)$, then r is the identity function. The function r is a scalar function that enables to compute the aggregation state s' without scanning the base dataset 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 introduce below the notion of a derivable set of an aggregation state s to define the set of aggregation states that can share the result of s.

Definition 3. *Let s be an aggregation state of a UDAF. The derivable set of s, denoted $D(s)$, is defined as follows: $D(s) = \{s' | s' \text{ shares } s\}$.*

The derivable set has the following two properties:

- **Transitive property.** Let s, s' and s'' be three aggregation states. If $s'' \in D(s')$ and $s' \in D(s)$, then we have $s'' \in D(s)$.
- **Symmetric property when reusing function is injective.** Let s and s' be two aggregation states and $s' \in D(s)$. Then, there exists a function r such that $s'(X) = r \circ s(X)$. In this case, if r is an injection, then we have $r^{-1} \circ s'(X) = s(X)$, such that $s \in D(s')$.

We use a derivable set of aggregation states to define the following decision problem underlying the issue of sharing aggregation states.

Problem 1. *Given two aggregation states s' and s, the sharing problem, noted share(s', s), is to decide whether $s' \in D(s)$?*

As stated by the following theorem, it is not possible to solve the sharing problem in a general setting.

Theorem 2. *The problem share(s', s) is undecidable.*

The proof of this theorem (detailed below), based on Rice's Theorem [HMU06], shows that the property of whether an aggregation state belongs to a derivable set is non-trivial and semantic.

Proof. Let AGG be the set of all aggregation states, which contains infinitive elements because of infinitive scalar functions and infinitive $\sum_{\oplus}$ functions for aggregation states. Let s be any aggregation state in AGG, then we prove below $D(s) \neq \emptyset$ and $D(s) \neq AGG$.

$D(s) \neq \emptyset$ since $D(s)$ contains at least one element s.

We explain $D(s) \neq AGG$ as follows. Let s and s' be two aggregation states. Assuming $s' \in D(s)$, then $s'(X) = r \circ s(X)$. W.l.o.g, let X_1 be an input multiset, and $s(X_1) = a$ and $s'(X_1) = b$. Then $r(a) = b$. Assuming X_2 is an another input multiset and $s(X_2) = a$. In this case, $s'(X_2)$ must equal to b otherwise r does not exist. In fact, in order to have $s' \in D(s)$, we have s' and s at least must satisfy $\forall X_1, X_2$, if $s(X_1) = s(X_2)$ then $s'(X_1) = s'(X_2)$. However, there is no guarantee for arbitrary elements s and s' in AGG to have this property. Such that, $D(s) \neq AGG$.

Finally, we have $\forall s \in AGG$, $D(s) \neq AGG$ and $D(s) \neq \emptyset$, which infers derivable set is a non-trivial property. Furthermore, it is straightforward to see $\forall X, s'(X) = s''(X)$ and if $s' \in D(s)$, then $s'' \in D(s)$. Then according to the Rice's Theorem [HMU06], the sharing problem share(s',s) is undecidable. □

3.3 Practical sharing framework

Since the sharing problem is undecidable in a general context, we deal with it for restricted aggregation functions that are widely used in practice.

We observe that a practical sharing framework of UDAFs should provide the following functionalities:

1. The UDAF framework can determine whether a practical aggregation state can share another practical aggregation state.
2. The UDAF framework should be able to generate partial aggregations of UDAFs.
3. Knowing that the interface for implementing UDAFs in DBMSs or data analytic systems provides a flexible approach for users to analyze data. Then, a qualified sharing framework of UDAFs should also allow users to create their UDAFs flexibly.

According to the above requirements, we design an expression model of aggregation functions, where we automatically generate a canonical form and partial aggregations of UDAFs (see a general discussion and related algorithms in Section 5.3). More specifically, we design three sets of functions, primitive scalar functions, primitive binary functions and primitive aggregation functions which contains restricted elements. Then, end users are able to create expressions of UDAFs arbitrarily by using elements from these sets. The three sets of primitive functions are proposed as follows (c.f., Table 3.1):

- *Primitive scalar functions.* The *PS* (primitive scalar) class contains six types of functions: constant, identity, linear, power, logarithmic and exponential functions.
- *Primitive binary functions.* The *PB* (primitive binary) class contains the following binary functions: addition $+$, subtraction $-$, multiplication $\times$, division $/$ and exponentiation $\wedge$.
- *Primitive aggregate functions.* The *PA* (primitive aggregate functions) class contains two functions: summation $\sum$ and product $\prod$.

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

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

TABLE 3.1: Classes of functions supported by SUDAF.

Complex scalar functions: PS° and $PS^\odot$. 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 3.1). The length of $g(x)$, denoted $|g|$, gives the number of primitive functions used in the definition of $g(x)$. For example, if $g(x) = h_l \circ ... \circ h_1(x)$, with $h_j \in PS$, then $|g| = l$. In addition, 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 3.1.

Supported UDAF. SUDAF also allows using the composition operator $\circ$ between scalar functions and primitive aggregate functions to define UDAFs. 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 3.1. Moreover, more complex UDAFs 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 3.1.

Scope of SUDAF. SUDAF restricts the set of UDAFs that can be declared to the classes presented in Table 3.1. We shall show in the next section that this restraint enables us to deal with the undecidability of the sharing problem. However, this restriction does not hamper the usability of SUDAF in real world applications since the proposed framework covers a wide range of aggregation functions such as the classes of power mean functions, arbitrary central moments [onl19a], arbitrary standardized moments [onl19b] and other multivariate aggregations [1] such as covariance and correlation. SUDAF supports also cofactor aggregates [SOC16] used in optimizing batch gradient descent to train least square regression model. Although holistic aggregation functions, e.g., the median, cannot be expressed in our framework, aggregation functions used in their approximation algorithms, e.g., the moment sketch [GDT+18], are supported by SUDAF.

[1] Multivariate aggregations can be seen as a combination of several uni-variate aggregations, each of which is expressed using functions in Table 3.1. 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 $\cdot$.

Mapping SUDAF functions into canonical forms Similarly, we can obtain a generic form of SUDAF functions, $\alpha(X) = T'((f'_k \circ \sum_{\oplus_k} \circ f_k(x_i)) \odot_{k-1} ... \odot_1 (f'_1 \circ \sum_{\oplus_1} \circ f_1(x_i)))$, where f_j, f'_j, for $j \in [1, ..., k]$, are complex scalar functions from $PS^{\odot}$, and $\sum_{\oplus_j}$ are one of the two primitive aggregation functions of the set $PA = \{\sum, \prod\}$, and $\odot_j, j < k$, are primitive binary functions from PB. Given such a function $\alpha(X) \in PA^{\odot}$, a canonical form $\text{canonical}(\alpha) = (F, \oplus, T)$ is derived from the general expression of α as follows:

- $F = (f_1, \ldots, f_k)$;
- $\oplus = (\oplus_1, \ldots, \oplus_k)$ and
- $T = T'\big((f'_1 \circ \underbrace{\sum_{\oplus_1} \circ f_1}_{s_1}) \odot_1 \ldots \odot_{k-1} (f'_k \circ \underbrace{\sum_{\oplus_k} \circ f_k}_{s_k})\big)$

The aggregation states of a UDAF α, according to its SUDAF canonical form $\text{canonical}(\alpha)$, are defined as follows: $s_j(X) = \sum_{\oplus_j} f_j(x_i)$, for $j \in [1, \ldots, k]$.

For instance, aggregations in Table 2.1 can be defined in SUDAF using their expressions in the second column in Table 2.1 (the composition operator is optional from users' side because SUDAF can automatically insert it in an expression). SUDAF generates their canonical forms from their expressions and factors out partial aggregation states, which are the s_i elements appearing in the T component in their canonical forms in Table 2.1.

To be summarized, SUDAF contains a limited number of functions for the sets PS, PB and PA in the expression model of UDAFs (see more details in Section 5.3), and it can automatically generate partial aggregations from expressions of UDAFs. The restriction of SUDAF functions does not hamper the flexibility of using SUDAF to create practical UDAFs. However, as we shall explain in the next section, this restriction can help solve the sharing problem.

3.4 Dealing with the sharing problem in SUDAF

In the previous section, we present that SUDAF can automatically generate a canonical form for a UDAF α given in a mathematical expression, and aggregation states of α can be straightforwardly obtained. It should be noteworthy that, SUDAF captures primitive functions (either aggregations or scalar functions) used in every aggregation states of α. As we shall discuss in this section, such kinds of semantics, i.e., what primitive functions used in aggregation states, can help deal with the sharing problem.

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 3.10). Therefore, we split the *sharing problem* $\text{share}(s_1, s_2)$ into four main cases depending on whether f_1 and f_2 are injections or even functions. The studied cases are presented in Table 3.2. Our main results provide a full characterization for the first three cases in Table 3.2. Specifically, we provide complete conditions, presented in Theorem 3, for the first two cases in Table 3.2, and we then reduce the third case to the second case in Table 3.2. We also propose an incomplete solution to deal with the fourth case in Table 3.2.

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 3)
2	-	Injective	Case 2 of Theorem 3
3	Even	Even	Case 2 of Theorem 3
4	Neither injective nor even	Neither injective nor even	Splitting rules (SR)

TABLE 3.2: Cases analysis of the sharing problem in SUDAF.

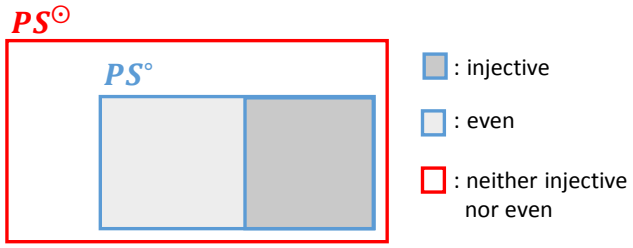

FIGURE 3.10: Injective and even functions in PS° and $PS^{\odot}$.

In the sequel, we use the function $sgn(x)$ defined as follows:

$$sgn(x) = \begin{cases} 1, x > 0; \\ 0, x = 0; \\ -1, x < 0. \end{cases}$$

3.4.1 Sharing conditions

We propose the following theorem to solve the *sharing problem* share(s_1, s_2), where f_1 is injective and f_2 is non-injective (corresponding to the case 1 in Table 3.2), and f_2 is injective (corresponding to the case 2 in Table 3.2).

Theorem 3. *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 $s_1(X) = r_{12} \circ s_2(X)$ if and only if 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 $r(x) = f_1 \circ f_2^{-1}(x)$.

We illustrate how Theorem 3 can be applied to deal with the *sharing problem* share(s_1, s_2) in the following example.

Example 2. *Let X be a multiset of rational values, i.e., $X \in \mathcal{M}(\mathbb{Q})$, and s_1 and s_2 are two aggregation states that are computed with X as input.*

- *Case 1. Given $s_1(X) = \sum 3x_i$ and $s_2(X) = \sum x_i^2$. In this case $f_1(x) = 3x$ and $f_2(x) = x^2$, then we have $f_1(x)$ is an injection and $f_2(x)$ is not an injection. According to the case 1 of Theorem 3, the answer to the problem* share*(s_1, s_2) is* no*.*

- *Case 2.1. Given $s_1(X) = \sum ln(x_i^2)$ and $s_2(X) = \sum log(x_i)$. In this case $f_1(x) = ln(x^2)$ and $f_2(x) = log(x)$, then we have both $f_1(x)$ and $f_2(x)$ are injections. Since $f_2(x)$ is an injection and $(\sum_{\oplus_1} = \sum_{\oplus_2} = \sum)$, case 2.1 of Theorem 3 is identified to be used. We have $f_1 \circ f_2^{-1}(x) = (ln4)x$, which satisfies the condition in case 2.1 of Theorem 3. Then, the corresponding answer to the problem* share*(s_1, s_2) is* yes*. Moreover, the case 2.1 of Theorem 3 also states that $r(x) = (ln4)x$ is the reusing function for $s_1(X) = r \circ s_2(X)$.*

- *Case 2.2. Given $s_1(X) = \sum 3x_i$ and $s_2(X) = \prod 2^{x_i}$. In this case $f_1(x) = 3x$ and $f_2(x) = 2^x$, then we have both $f_1(x)$ and $f_2(x)$ are injections. Since $f_2(x)$ is an injection and $(\sum_{\oplus_1} = \sum, \sum_{\oplus_2} = \prod)$, the case 2.2 of Theorem 3 is identified to be used. We have $f_1 \circ f_2^{-1}(x) = 3log(x)$, which satisfies the condition in case 2.2 of Theorem 3. Then, the corresponding answer to the problem* share*(s_1, s_2) is* yes*. Moreover, the case 2.2 of Theorem 3 also states that $r(x) = 3log(x)$ is the reusing function for $s_1(X) = r \circ s_2(X)$.*

- *Case 2.3. Given $s_1(X) = \prod x_i^2$ and $s_2(X) = \sum ln(x_i)$. In this case $f_1(x) = x^2$ and $f_2(x) = ln(x)$, then we have both $f_1(x)$ and $f_2(x)$ are injections. Since $f_2(x)$ is an injection and $(\sum_{\oplus_1} = \prod, \sum_{\oplus_2} = \sum)$, the case 2.3 of Theorem 3 is identified to be used. We have $f_1 \circ f_2^{-1}(x) = e^{2x}$, which satisfies the condition in case 2.3 of Theorem 3. Then, the corresponding answer to the problem* share*(s_1, s_2) is* yes*. Moreover, the case 2.3 of Theorem 3 also states that $r(x) = e^{2x}$ is the reusing function for $s_1(X) = r \circ s_2(X)$.*

- *Case 2.4. Given $s_1(X) = \prod 2^{x_i}$ and $s_2(X) = \prod e^{3x_i}$. In this case $f_1(x) = 2^x$ and $f_2(x) = e^{3x}$, then we have both $f_1(x)$ and $f_2(x)$ are injections. Since $f_2(x)$ is an injection and $(\sum_{\oplus_1} = \sum_{\oplus_2} = \prod)$, the case 2.4 of Theorem 3 is identified to be used. We have $f_1 \circ f_2^{-1}(x) = x^{\frac{1}{3log(e)}}$, which satisfies the condition in case 2.4 of Theorem 3. Then, the corresponding answer to the problem* share*(s_1, s_2) is* yes*. Moreover, the case 2.4 of Theorem 3 also states that $r(x) = x^{\frac{1}{3log(e)}}$ is the reusing function for $s_1(X) = r \circ s_2(X)$.*

The case 1 of Theorem 3 states that, given two aggregations 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 identify function when $s_1 = s_2$, it is not possible to find a computable function r_{12} such that $s_1(X) = r_{12} \circ s_2(X)$. A proof is given below.

Proof. **(Case 1 of Theorem 3)** We prove this case by contradiction.
Assuming r_{12} exists s.t. $s_1(X) = r_{12} \circ s_2(X)$. Then for any two multisets X and Y, we have:

$$\text{if } s_2(X) = s_2(Y), \text{ then } s_1(X) = s_1(Y). \tag{3.2}$$

Let f_2^{-q} be a quasi-inverse function[2] of f_2. Assume two multisets $X = \{\{x_1, ..., x_n\}\}$ and $Y = \{\{y_1, y_2\}\}$ with $y_1 = f_2^{-q}(f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1}))$ and $y_2 = x_n$. Therefore, we have

$$\begin{aligned} s_2(Y) &= f_2(y_1) \oplus_2 f_2(y_2) \\ &= f_2(f_2^{-q}(f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1})) \oplus_2 f_2(x_n) \\ &= f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1}) \oplus_2 f_2(x_n) \\ &= s_2(X). \\ \therefore s_1(Y) &= s_1(X) \text{ (the condition in equation (3.2))} \end{aligned}$$

Since f_2 is not an injection, then it can have several quasi-inverse functions. Let $f_2^{-q'}$ be another quasi-inverse function of f_2, which is different from f_2^{-q}, such that $y_1' = f_2^{-q'}(f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1})) \neq y_1$. Let $Y' = \{\{y_1', y_2\}\}$, then, we have:

$$\begin{aligned} s_2(Y') &= f_2(y_1') \oplus_2 f_2(y_2) \\ &= f_2(f_2^{-q'}(f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1})) \oplus_2 f_2(x_n) \\ &= f_2(x_1) \oplus_2 ... \oplus_2 f_2(x_{n-1}) \oplus_2 f_2(x_n) \\ &= s_2(X). \\ \therefore\ s_2(Y') &= s_2(Y), \\ \therefore\ s_1(Y) &= s_1(Y') \text{ (the condition in equation (3.2))} \end{aligned}$$

As explained below, this is not possible. On another side, since f_1 is an injection, we have $f_1(y_1) \neq f_1(y_1')$ (because $y_1 \neq y_1'$). Then, for $\sum_{\oplus_1} \in PA$, there exists $f_1(y_2)$, such that $f_1(y_1) \oplus_1 f_1(y_2) \neq f_1(y_1') \oplus_1 f_1(y_2)$. Therefore, $s_1(Y) \neq s_1(Y')$ which contradicts the fact that $s_1(Y) = s_1(Y')$. Such that r_{12} does not exist. □

The case 2 of Theorem 3 provides necessary and sufficient conditions to characterize solutions of the share(s_1, s_2) problem 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. Specifically, we found the form of $f_1 \circ f_2^{-1}(x)$ by using Cauchy's functional equation [Sma07, onl18] in the case 2.1, which is a complete condition for solving the sharing problem in the case 2.1. The other three sub-cases follows the same structure of the case 2.1. The proof is detailed below.

Proof. **(Case 2.1 of Theorem 3)**
(Sufficiency) Assume that $f_1 \circ f_2^{-1}(x) = ax$. Then, we have:

$$\begin{aligned} f_1 \circ f_2^{-1} \circ s_2(X) &= a(\sum f_2(x_i)) \\ &= \sum a(f_2(x_i)) \\ &= \sum f_1 \circ f_2^{-1} \circ f_2(x_i) \\ &= s_1(X). \end{aligned}$$

Hence, taking $r_{12}(x) = f_1 \circ f_2^{-1}(x) = ax$, we have $s_1(X) = r_{12} \circ s_2(X)$.

(Necessity) Assume that there exists a function r_{12} s.t. $s_1(X) = r_{12} \circ s_2(X)$. Then, we have $\sum f_1(x_i) = r_{12} \circ \sum f_2(x_i)$. Hence, we can derive that $\sum f_1 \circ f_2^{-1}(x_i) = r_{12} \circ \sum f_2 \circ f_2^{-1}(x_i)$. Then, $\sum f_1 \circ f_2^{-1}(x_i) = r_{12} \circ \sum x_i$. If we note $g(x) = f_1 \circ f_2^{-1}(x)$, we obtain

[2] Every function has a quasi-inverse function by the Axiom of Choice. If $g(x)$ is a quasi-inverse of $f(x)$, then $f \circ g \circ f(x) = f(x)$ or $f \circ g(x) = x$.

$\sum g(x_i) = r_{12} \circ \sum x_i$. Let $\{\{x_1, ..., x_n\}\}$ and $\{\{y_1, y_2\}\}$ be two multisets with $y_1 = x_1 + ... + x_{n-1}$ and $y_2 = x_n$. Then, we have:

$$g(x_1) + ... + g(x_n) = r_{12}(x_1 + ... + x_n), \text{ and} \tag{3.3}$$
$$r_{12}(y_1 + y_2) = g(y_1) + g(y_2). \tag{3.4}$$

Knowing that $x_1 + ... + x_n = y_1 + y_2$, and hence $r_{12}(x_1 + ... + x_n) = r_{12}(y_1 + y_2)$, and from equations *(3.3)* and *(3.4)*, we derive $g(x_1) + ... + g(x_n) = g(y_1) + g(y_2)$. Then, from $y_1 = x_1 + ... + x_{n-1}$, we obtain

$$g(x_1) + ... + g(x_{n-1}) = g(x_1 + ... + x_{n-1}). \tag{3.5}$$

Note that, equation *(3.5)* is a Cauchy's functional equation [Sma07, onl18]. This implies that $g(x)$ is an additive function having the following form:

$$g(x) = ax, x \in \mathbb{Q}, a \in \mathbb{Q}_{\neq 0}. \tag{3.6}$$

From equation (3.6) and $\sum g(x_i) = r_{12} \circ \sum x_i$, we have $g(x) = r(x)$. Such that,

$$f_1 \circ f_2^{-1}(x) = r_{12}(x) = ax, x \in \mathbb{Q}, a \in \mathbb{Q}_{\neq 0}.$$

□

Proof. **(Case 2.2 of Theorem 3)**
(Sufficiency) Assume that $f_1 \circ f_2^{-1}(x) = a(log_b|x|)$. Then,

$$\begin{aligned} f_1 \circ f_2^{-1} \circ s_2(X) &= a(log_b|(\prod f_2(x_i))|) \\ &= \sum a(log_b|f_2(x_i)|) \\ &= \sum f_1 \circ f_2^{-1} \circ f_2(x_i) \\ &= s_1(X). \end{aligned}$$

Hence, if we take $r_{12}(x) = f_1 \circ f_2^{-1}(x) = a(log_b|x|)$, we have $s_1(X) = r_{12} \circ s_2(X)$.

(Necessity) Assume that there exists a function r_{12} s.t. $s_1(X) = r_{12} \circ s_2(X)$. Then, we have $\sum f_1(x_i) = r_{12} \circ \prod f_2(x_i)$ and hence $\sum f_1 \circ f_2^{-1}(x_i) = r_{12} \circ \prod x_i$. If we note $g(x) = f_1 \circ f_2^{-1}(x)$, we obtain $\sum g(x_i) = r_{12} \circ \prod x_i$. Let $\{\{x_1, ..., x_n\}\}$ and $\{\{y_1, y_2\}\}$ be two multisets with $x_1 \times ... \times x_{n-1} = y_1$ and $x_n = y_2$. Then, we have

$$r(x_1 \times ... \times x_n) = g(x_1) + ... + g(x_n), \text{ and} \tag{3.7}$$
$$r(y_1 \times y_2) = g(y_1) + g(y_2). \tag{3.8}$$

Knowing that $y_1 \times y_2 = x_1 \times ... \times x_n$, and hence $r(x_1 \times ... \times x_n) = r(y_1 \times y_2)$, and from equation *(3.7)* and *(3.8)*, we derive $g(x_1) + ... + g(x_n) = g(y_1) + g(y_2)$. Then from $y_1 = x_1 \times ... \times x_{n-1}$, we obtain

$$g(x_1 \times ... \times x_{n-1}) = g(x_1) + ... + g(x_{n-1}). \tag{3.9}$$

Equation (3.9) can be transformed to a Cauchy's functional equation, which implies: $g(x) = a(log_b(x)), x \in \mathbb{Q}_{>0}, b \in \mathbb{Q}_{>0,\neq 1}, a \in \mathbb{Q}_{\neq 0}$. We explain this in details below.

We can derive from equation (3.9) that $g(x)$ does not define on 0, otherwise $g(x) = 0$, which is a contradiction to the condition of a non-constant f_1. Then we have $g(x)$ can be

defined on either $\mathbb{Q}_{>0}$ or $\mathbb{Q}_{<0}$. For $x > 0$, let $x = b^u, u \in \mathbb{Q}$, and $h(u) = g(b^u), b \in \mathbb{Q}_{>0,\neq 1}$, then we have:

$$\begin{aligned} h(u_1 + ... + u_{n-1}) =& g(b^{u_1+...+u_{n-1}}) \\ =& g(b^{u_1} \times ... \times b^{u_{n-1}}) \\ =& g(b^{u_1}) + ... + g(b^{u_{n-1}}) \\ =& h(u_1) + ... + h(u_{n-1}). \end{aligned}$$

Then, according to Cauchy's functional equation [Sma07, onl18], we have $h(u) = h(1)u$, with a constant $h(1) \in \mathbb{Q}_{\neq 0}$. Then $g(b^u) = h(1)u$ and $u = log_b x$, such that $g(x) = a(log_b x), x \in \mathbb{Q}_{>0}, b \in \mathbb{Q}_{>0,\neq 1}, a = h(1) \in \mathbb{Q}_{\neq 0}$. From equation (3.9), we can also have $g(1) = 0$ and $g(-1) = 0$, such that $g(x) = g(-x)$. Then for $x < 0$, we have $g(x) = a(log_b(-x))$. Therefore, with constants $a \in \mathbb{Q}_{\neq 0}$ and $b_{>0,\neq 1}$, we have:

$$g(x) = a(log_b|x|), x \in \mathbb{Q}_{\neq 0} \tag{3.10}$$

From equation (3.10) and $\sum g(x_i) = r_{12} \circ \prod x_i$, we have $g(x) = r(x)$. Such that

$$f_1 \circ f_2^{-1}(x) = r_{12}(x) = a(log_b|x|), x \in \mathbb{Q}_{\neq 0}, b \in \mathbb{Q}_{>0,\neq 1}, a \in \mathbb{Q}_{\neq 0}.$$

□

Proof. **(Case 2.3 of Theorem 3)**

(Sufficiency) Assume that $f_1 \circ f_2^{-1}(x) = b^{ax}$. Then, we have:

$$\begin{aligned} f_1 \circ f_2^{-1} \circ s_2(X) &= b^{a(\sum f_2(x_i))} \\ &= \prod b^{a(f_2(x_i))} \\ &= \prod f_1 \circ f_2^{-1} \circ f_2(x_i) \\ &= s_1(X). \end{aligned}$$

Then, if we take $r_{12}(x) = f_1 \circ f_2^{-1}(x) = b^{ax}$, we have $s_1(X) = r_{12} \circ s_2(X)$.

(Necessity) Assume that there exists a function r_{12} s.t. $s_1(X) = r_{12} \circ s_2(X)$. Then, we have $\prod f_1(x_i) = r_{12} \circ \sum f_2(x_i)$ and hence $\prod f_1 \circ f_2^{-1}(x_i) = r_{12} \circ \sum x_i$. If we note $g(x) = f_1 \circ f_2^{-1}(x)$, we obtain $\prod g(x_i) = r_{12} \circ \sum x_i$. Let $\{\{x_1, ..., x_n\}\}$ and $\{\{y_1, y_2\}\}$ be two multisets with $x_1 + ... + x_{n-1} = y_1$, $x_n = y_2$ and $g(x_n) \neq 0$. Then, we have

$$r(x_1 + ... + x_n) = g(x_1) \times ... \times g(x_n), \text{ and} \tag{3.11}$$

$$r(y_1 + y_2) = g(y_1) \times g(y_2). \tag{3.12}$$

Knowing that $y_1 + y_2 = x_1 + ... + x_n$, and hence $r(x_1 + ... + x_n) = r(y_1 + y_2)$, and from equation *(3.11)* and *(3.12)*, we derive $g(x_1) \times ... \times g(x_n) = g(y_1) \times g(y_2)$. Then from $x_1 + ... + x_{n-1} = y_1$, we obtain

$$g(x_1 + ... + x_{n-1}) = g(x_1) \times ... \times g(x_{n-1}). \tag{3.13}$$

Equation *(3.13)* can be transformed to a Cauchy's functional equation, which implies: $g(x) = b^{ax}, x \in \mathbb{Q}, b \in \mathbb{Q}_{>0,\neq 1}, a \in \mathbb{Q}_{\neq 0}$. We explain this in details below.

We can derive from equation *(3.13)* that $\forall x \in \mathbb{Q}, g(x) > 0$. We have $g(x) = g(\frac{x}{2} + \frac{x}{2}) = (g(\frac{x}{2}))^2$, which implies $g(x) \geqslant 0$. We can also have $\forall x \in \mathbb{Q}, g(x) \neq 0$. W.l.o.g, let $n = 3, x_1 = 0, x_2 = x$, then $g(x) = g(0) \times g(x)$, which implies $g(0) \neq 0$, otherwise $g(x) = 0$ and $g(x)$ is a constant function contradicting to a non-constant f_1. Then, we

have $g(0) = 1$ by $g(0) = (g(0))^2$. Moreover, $g(x) \times g(-x) = g(x-x) = g(0) = 1$, such that we have $\forall x \in \mathbb{Q}, g(x) \neq 0$. Let $h(x) = log_b(g(x)), x \in \mathbb{Q}, b \in \mathbb{Q}_{>0,\neq 1}$, then we have

$$\begin{aligned} h(x_1 + ... + x_{n-1}) =& log_b(g(x_1 + ... + x_{n-1})) \\ =& log_b(g(x_1) \times ... \times g(x_{n-1})) \\ =& log_b(g(x_1)) + ... + log_b(g(x_{n-1})) \\ =& h(x_1) + ... + h(x_{n-1}). \end{aligned}$$

Then, according to Cauchy's functional equation [Sma07,onl18], we have $h(x) = h(1)x, x \in \mathbb{Q}, h(1) \in \mathbb{Q}_{\neq 0}$. Such that,

$$g(x) = b^{ax}, x \in \mathbb{Q}, b \in \mathbb{Q}_{>0,\neq 1}, a = h(1) \in \mathbb{Q}_{\neq 0}. \tag{3.14}$$

From equation (3.14) and $\prod g(x_i) = r_{12} \circ \sum x_i$, we have $g(x) = r(x)$. Such that

$$f_1 \circ f_2^{-1}(x) = r_{12}(x) = b^{ax},\ x \in \mathbb{Q}, b \in \mathbb{Q}_{>0,\neq 1}, a \in \mathbb{Q}_{\neq 0}.$$

□

Proof. **(Case 2.4 of Theorem 3)**
(Sufficiency) Assume $f_1 \circ f_2^{-1}(-1) = 1$ and $f_1 \circ f_2^{-1}(x) = |x|^a$. Then we have,

$$\begin{aligned} r_{12} \circ \prod f_2(x_i) &= |x|^a \circ \prod f_2(x_i) \\ &= (|\prod f_2(x_i)|)^a \\ &= (\prod |f_2(x_i)|)^a \\ &= \prod (|f_2(x_i)|)^a \\ &= \prod |x|^a \circ f_2(x_i) \\ &= \prod f_1 \circ f_2^{-1} \circ f_2(x_i) \\ &= \prod_{i=1}^{n} f_1(x_i) \\ &= s_1(X). \end{aligned}$$

Assume $f_1 \circ f_2^{-1}(-1) = -1$ and $f_1 \circ f_2^{-1}(x) = sgn(x) \times |x|^a$. Then we have,

$$\begin{aligned} r_{12} \circ \prod f_2(x_i) &= sgn(\prod f_2(x_i)) \times |\prod f_2(x_i)|^a \\ &= (\prod sgn(f_2(x_i))) \times \prod |f_2(x_i)|^a \\ &= \prod sgn(f_2(x_i)) \times |f_2(x_i)|^a \\ &= \prod f_1 \circ f_2^{-1} \circ f_2(x_i) \\ &= \prod_{i=1}^{n} f_1(x_i) \\ &= s_1(X). \end{aligned}$$

(Necessity) Assume that there exists a function r_{12} s.t. $s_1(X) = r_{12} \circ s_2(X)$. Then, we have $\prod f_1(x_i) = r_{12} \circ \prod f_2(x_i)$ and hence $\prod f_1 \circ f_2^{-1}(x_i) = r_{12} \circ \prod x_i$. If we note $g(x) = f_1 \circ f_2^{-1}(x)$, we obtain $\prod g(x_i) = r_{12} \circ \prod x_i$. Let $\{\{x_1, ..., x_n\}\}$ and $\{\{y_1, y_2\}\}$ be two multisets with $x_1 \times ... \times x_{n-1} = y_1$, $x_n = y_2$ and $g(x_n) \neq 0$. Then, we have

$$r(x_1 \times ... \times x_n) = g(x_1) \times ... \times g(x_n), \text{ and} \tag{3.15}$$

$$r(y_1 \times y_2) = g(y_1) \times g(y_2). \tag{3.16}$$

Knowing that $x_1 \times ... \times x_n = y_1 \times y_n$, and hence $r(x_1 \times ... \times x_n) = r(y_1 \times y_2)$, and from equation *(3.15)* and *(3.16)*, we derive $g(x_1) \times ... \times g(x_n) = g(y_1) \times g(y_2)$. Then from $x_1 \times ... \times x_{n-1} = y_1$, we obtain

$$g(x_1 \times ... \times x_{n-1}) = g(x_1) \times ... \times g(x_{n-1}). \tag{3.17}$$

Equation *(3.17)* can be transformed to a Cauchy's functional equation [Sma07, onl18], which implies $g(x) = x^a$, for $x > 0$. We explain this below in details.

When $x > 0$, we derive that $g(x) > 0$. From equation (3.17) we can have $g(x) = g(1) \times g(x)$, then $g(1) \neq 0$, otherwise g is a constant function contradicting to non-constant f_1. Then we have $g(1) = 1$ because of $g(1) = (g(1))^2$. Moreover, $\forall x \neq 0$, we have $g(x) \times g(\frac{1}{x}) = g(x \times \frac{1}{x}) = g(1) = 1$, then $g(x) \neq 0$. Furthermore, we have $g(x^2) = g^2(x)$. Such that, we have $\forall x \in \mathbb{Q}_{>0}, g(x) > 0$. Let $x = e^u, u \in \mathbb{Q}$ and $h(u) = ln(g(e^u))$, then we have

$$\begin{aligned} h(u_1 + ... + u_{n-1}) =& ln(g(e^{u_1+...+u_{n-1}})) \\ =& ln(g(e^{u_1} \times ... \times e^{u_{n-1}})) \\ =& ln(g(e^{u_1}) \times ... \times g(e^{u_{n-1}})) \\ =& ln(g(e^{u_1})) + ... + ln(g(e^{u_{n-1}})) \\ =& h(u_1) + ... + h(u_{n-1}). \end{aligned}$$

Then, according to Cauchy's functional equation [Sma07,onl18], we have $h(u) = h(1)u, u \in \mathbb{Q}, h(1) \in \mathbb{Q}_{\neq 0}$, then $ln(g(e^u)) = h(1)u$. Such that, we have

$$g(x) = x^a, x \in \mathbb{Q}_{>0}, a \in \mathbb{Q}_{\neq 0}.$$

When $x < 0$, from equation (3.17) we have $g(x) = g(-1)g(-x) = g(-1)(-x)^a$. Moreover, we have $g(1) = g(-1) \times g(-1)$, such that we have either $g(-1) = 1$ or $g(-1) = -1$. Then, we have for $x < 0$:

(i) when $g(-1) = 1$ and $x < 0$, $g(x) = (-x)^a$;

(ii) when $g(-1) = -1$ and $x < 0$, $g(x) = -(-x)^a$.

Therefore, we have the function $g(x)$:

(i) when $g(-1) = 1, g(x) = |x|^a, x \neq 0$;

(ii) when $g(-1) = -1, g(x) = sgn(x) \times |x|^a, x \neq 0$.

From $\prod g(x) = r_{12} \circ \prod x$ and equation (3.17), we have $g(x) = r(x)$. □

3.4.2 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 3.2). As depicted in Figure 3.10, 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 3 can be applied in this setting. We denote $U_X = \{u_x = |x| | x \in X\}$. 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 $s_2(X)$ if and only if $s_1(U_X)$ 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 3 can be exploited to solve the sharing problem in this context.

Example 3. *Let X be a multiset of rational values, i.e., $X \in \mathcal{M}(\mathbb{Q})$. Given $s_1(X) = \sum ln(x_i^2)$ and $s_2(X) = \sum log(x_i^4)$. In this case $f_1(x) = ln(x^2)$ and $f_2(x) = log(x^4)$, then we have both $f_1(x)$ and $f_2(x)$ are even functions. In the sequel, we consider $f_1(x)$ and $f_2(x)$ over the positive domain, where both $f_1(x)$ and $f_2(x)$ are injections. Since $(\sum_{\oplus_1} = \sum_{\oplus_2} = \sum)$, the case 2.1 of Theorem 3 is identified to be used. We have $f_1 \circ f_2^{-1}(x) = (\frac{ln2}{2})x$, which satisfies the condition in case 2.1 of Theorem 3. Then, the corresponding answer to the problem share(s_1, s_2) is yes. Moreover, the case 2.1 of Theorem 3 also states that $r(x) = (\frac{ln2}{2})x$ is the reusing function for $s_1(X) = r \circ s_2(X)$.*

3.4.3 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 3.2). As depicted in Figure 3.10, 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 [3], 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 3.

Example 4. *Let X be a multiset of rational values, i.e., $X \in \mathcal{M}(\mathbb{Q})$. Given $s_1(X) = \sum x_i^2 + ln(x_i^4)$ and $s_2(X) = \sum 3x_i^2 - log(x_i^3)$. In this case $f_1(x) = x^2 + ln(x^4)$ and $f_2(x) = 3x^2 - log(x^3)$, then we have both $f_1(x)$ and $f_2(x)$ are neither injections nor even functions. As a consequence of this step, Theorem 3 cannot be directly used in this case. However, we can apply splitting rules to transform the expression of $s_1(X)$ and $s_2(X)$. We show the transformation of aggregate expression trees of s_1 and s_2 in Figure 3.11. Consequently, both s_1 and s_2 are split into two aggregation states shown as follows,*

$$s_1(X) = s_1'(X) + s_1''(X), \text{ where } s_1'(X) = \sum x_i^2 \text{ and } s_1''(X) = \sum ln(x_i^4);$$
$$s_2(X) = s_2'(X) - s_2''(X), \text{ where } s_2'(X) = \sum 3x_i^2 \text{ and } s_2''(X) = \sum log(x_i^3).$$

Assuming that s_1 has been transformed using splitting rule 1, then we compute and cache $s_1'(X)$ and $s_1''(X)$, from which we compute $s_1(X)$. Then, given s_2, we also apply splitting rule 1 to transform s_2. Now, the problem share(s_1, s_2) has been reduced to the following problem, whether the cache of $s_1'(X)$ and $s_1''(X)$ can be reused to compute $s_2(X)$, which includes solving a sharing problem for s_1', that either share(s_1', s_2') or share(s_1', s_2''), and a sharing problem for s_1'', that either share(s_1'', s_2') or share(s_1'', s_2''). Note that, Theorem 3 can be directly used to solve these 4 problems. Specifically, according to the case 2.1 of Theorem 3, we have the answer to share(s_1', s_2') is yes, and the answer to share(s_1', s_2'') is yes. Therefore, s_2 can be computed from the cache $s_1'(X)$ and $s_1''(X)$ other than the base data.

[3] In practical implementation, we generate an expression tree for every aggregate function, which is denoted as aggregate expression tree (AET). Then, we apply transformation rules on AETs. See Section 5.3 for more details.

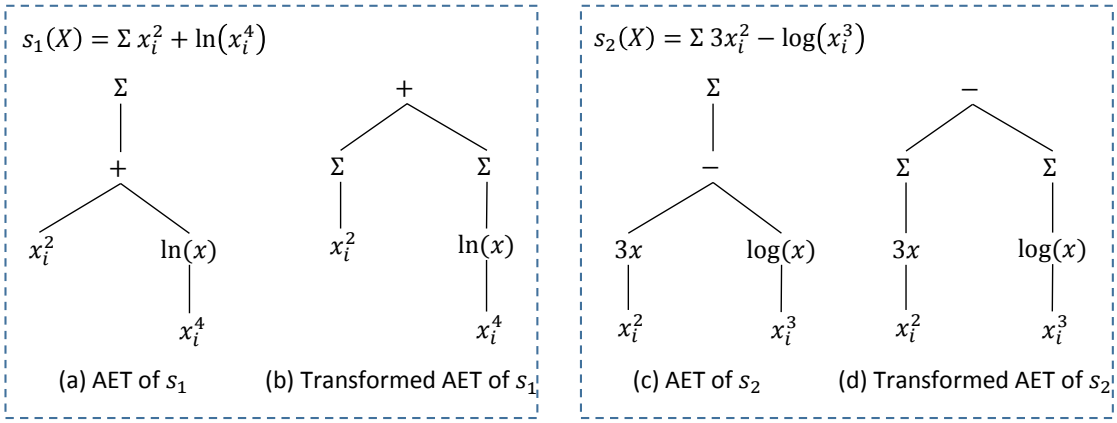

FIGURE 3.11: Transforming aggregate expression trees (AET) using splitting rules

3.5 Extension to multivariate functions

In Section 3.4, we propose the sharing conditions to deal with sharing problems for aggregation functions that are uni-variate functions, which are computed over one column in a table. In real-world data analysis, one also needs to compute aggregate functions that are multivariate, which aggregates data from several columns of a table. In this section, we briefly discuss the extension of sharing conditions to solve the sharing problem for multivariate functions.

The general idea is to extract from a multivariate aggregation function a set of aggregation states and reduce such a set of multivariate aggregation states to several uni-variate ones. We use the following two approaches to accomplish this goal,

- We check whether it is possible to apply splitting rules to obtain uni-variate aggregation states. For instance, given a multivariate aggregation state $s(X,Y) = \sum x_i^2 + ln(y_i)$, which aggregates values from two columns X and Y. In this case, $s(X,Y)$ can be split into two uni-variate aggregation states $\sum x_i^2$ and $\sum ln(y_i)$. Then, we can reuse the approach presented in Section 3.4 to respectively check whether there is an aggregation state in the cache that can be reused to compute $\sum x_i^2$, or $\sum ln(y_i)$. Note that, aggregation states are grouped based on their input column, i.e., $\sum x_i^2$ belongs to a group which is different from the group of $\sum ln(y_i)$ since they are computed over different columns, and we solve sharing problems for aggregation states that are from identical group.

- If it is impossible to apply splitting rules, we can annotate a scalar function of a multivariate aggregation state as an intermediate column. For instance, the covariance, $cov(X,Y) = \dfrac{\sum(x_i \times y_i)}{n} - \dfrac{\sum x_i \times \sum y_i}{n^2}$, has a multivariate aggregation state $s_1(X,Y) = \sum x_i \times y_i$. Then, $s_1(X,Y)$ can be annotated as a uni-variate aggregation over an abstract intermediate column Z, that $s(X,Y) = s(Z)$, where Z is the scalar product of X and Y, i.e., $Z = X \cdot Y$. As a consequence, if there is another aggregation state over Z, i.e., $s_2(Z)$, then we can reuse the approach in Section 3.4 to deal with the problem share(s_1, s_2) over Z. Note that, this approach requires checking that whether s_1 and s_2 are computed over an identical intermediate column, or s_1 and s_2 have the identical scalar function, which is expensive for a scalar function $f \in PS^{\odot} \setminus PS^{\circ}$, because f can contain addition and multiplication which is commutative and associative. While a practical and efficient method is to compare the shape and nodes of expression trees for scalar functions.

3.6 Summary

- In order to reduce data access during computing various UDAFs, we propose to cache and reuse aggregation states, which is an intermediate computation result of the canonical form of UDAFs. We formalize the caching and reusing problem as a sharing problem on aggregation states of UDAFs, and we show that it is undecidable in a general setting.
- We deal with the sharing problem in the context of practical aggregation functions. We present a practical UDAF framework SUDAF, which is an instance of the aggregation expression model. SUDAF also contains three classes of primitive functions, PS, PB and PA. One can use elements of them to create expressions of UDAFs. In each class of primitive functions, SUDAF only contains limit types of functions. We show that the limited scope of SUDAF functions covers a wide range of practical aggregations.
- We deal with the sharing problem for UDAFs in SUDAF. We characterize 4 cases based on scalar functions in aggregation states and identify complete conditions for sharing problems under the first two cases. Moreover, we show that the other two cases can be reduced to the first two cases.
- We discuss how to reduce multivariate aggregation states, which aggregates data over more than one column, to uni-variate ones, such that sharing conditions can be applied to solve sharing problems for multivariate aggregation states.

Chapter 4

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 3. Turning the conditions of Theorem 3 into an algorithm could be cumbersome because equivalent mathematical expressions may have different syntactic shapes.

Example 5. *Consider the problem whether* $s_1(X) = \sum 4x_i^2$ *shares* $s_2(X) = \sum(3x_i)^2$. *Using Theorem 3, 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 3, we have to check whether* $f_1 \circ f_2^{-1}(x) = ax$, *for some constant* a. *This is not an easy task, since this requires some 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 3.*

In addition, a straightforward implementation of Theorem 3 leads to redundant computations as illustrated below.

Example 6. *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 [ZT19] 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.

4.1 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, $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x)$ is a symbolic scalar function $sf_{\bar{p}}(x)$ with a sequence $\bar{p} = (p_l, ..., p_1)$ of parameters. Similarly, $|sf_{\bar{p}}| = l$.

Symbolic aggregation states. Intuitively, $\sum p_2 x^{p_1}$ is the symbolic representation of the aggregation state $\sum 3x^2$. In this case, $\sum p_2 x^{p_1}$ is called a symbolic (aggregation) state and we say that the concrete state $\sum 3x^2$ is an instance of the symbolic state $\sum p_2 x^{p_1}$. 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 $f(x) = x$.

Mapping an aggregation state to a unique symbolic state. Note that an aggregation state can be an instance of several symbolic states, which will become an issue when mapping an aggregation state to a precomputed sharing relationship of symbolic states. For example, the aggregation state $\sum x_i^a$, with $a = 1$, is an instance of the symbolic states $\sum x_i^p$ and $\sum x_i$. We define below the notion of *natural instances* of symbolic aggregation states, which enable us to associate any aggregation state to a unique symbolic state.

Definition 4. *An aggregation state $s(X)$ is a natural instance of a symbolic aggregation state $ss(X) = \sum_{\oplus} sf_{\bar{p}}(x_i)$ if and only if $\forall ss'(X) = \sum_{\oplus'} sf'_{\bar{p}'}(x_i) \in saggs_l(X)$ with $|sf'_{\bar{p}'}| \leqslant |sf_{\bar{p}}|$ and $ss'(X) \neq ss(X)$, $s(X)$ is not an instance of $ss'(X)$.*

In general, given an aggregation state $s = \sum_{\oplus} f(x_i)$, it is straightforward to compute the symbolic state ss such that s is a natural instance of ss: it is in general enough to take $ss = \sum_{\oplus} sf_{\bar{p}}(x_i)$ where $sf_{\bar{p}}$ is the symbolic function obtained from f by replacing each constant that appears in f with a fresh parameter. For example, aggregation states of the form $\sum ax_i$ and $\sum x_i^b$ with constants $a \neq 1$ and $b \neq 1$ are respectively natural instances of $\sum px_i$ and $\sum x_i^p$. Some particular cases must be treated cautiously because they require a preprocessing step before applying such a transformation. For example, aggregation states of the form $\sum a_2^{log_{a_1} x_i}$ with $a_2 = a_1$ are not natural instances of symbolic states of the form $\sum p_2^{log_{p_1} x_i}$ as it could be expected by applying the previous transformation rule but are instead natural instances of the symbolic state $\sum x_i$. Such cases are handled by applying a set of reduction rules, explained later in this section, on the scalar function of the aggregation state before generating the corresponding symbolic state. To continue with our previous example, the state $\sum a_2^{log_{a_1} x_i}$ is first rewritten into $\sum x_i$ and then translated into a symbolic state.

Given an aggregation state $s(X) = \sum_{\oplus} f(x_i)$, in order to compute the symbolic state ss such that s is a natural instance of ss, we reduce $|f|$ using reduction rules. The goal of this preprocessing step is to obtain a scalar function $f'(x)$ having the following two properties: (1) $f'(x) = f(x)$ and $sf'_{\bar{p}'}(x) \neq sf_{\bar{p}}(x)$; (2) $\nexists f''(x)$ such that $f''(x) = f(x)$,

$sf''_{\bar{p}'}(x) \neq sf_{\bar{p}}(x)$ and $|f''| < |f'|$. We exhaustively list reduction rules (RRs) for the cases $|f| = 1$ and $|f| = 2$ as follows, where a_1 and a_2 are constants used in primitive functions.

RR1: $a_1 x \rightarrow$ if $(a_1 = 1)\ x$.
RR2: $x^{a_1} \rightarrow$ if $(a_1 = 1)\ x$.
RR3: $a_2 x \circ a_1 x \rightarrow a_2 a_1 x$.
RR4: $a_2 x \circ x^{a_1} \rightarrow$ if $(a_2 = 1)\ x^{a_1}$, or if $(a_1 = 1)\ a_2 x$.
RR5: $a_2 x \circ log_{a_1} x \rightarrow$ if $(a_2 = 1)\ log_{a_1} x$.
RR6: $a_2 x \circ {a_1}^x \rightarrow$ if $(a_2 = 1)\ {a_1}^x$.
RR7: $x^{a_2} \circ a_1 x \rightarrow$ if $(a_1 > 0)\ a_1^{a_2} x \circ x^{a_2}$.
RR8: $x^{a_2} \circ x^{a_1} \rightarrow x^{a_2 a_1}$.
RR9: $x^{a_2} \circ a_1^x \rightarrow a_2^{a_2 x}$.
RR10: $log_{a_2} x \circ x^{a_1} \rightarrow$ if $(a_1 \neq 2k, k \in \mathbb{Z})\ a_1 x \circ log_{a_2} x$.
RR11: $log_{a_2} x \circ {a_1}^x \rightarrow (log_{a_2} a_1) x$.
RR12: ${a_2}^x \circ a_1 x \rightarrow a_2^{a_1 x}$.
RR13: ${a_2}^x \circ log_{a_1} x \rightarrow x^{1/log_{a_2} a_1}$.

We use the above rules to reduce the length of a scalar function in an aggregation state in the following example.

Example 7. *Given an aggregation state $\sum f(x_i)$ with the scalar function $f(x) = log_{a_2} x \circ x^{a_1}$ with constants a_2 and $a_1, a_1 \neq 2k, k \in \mathbb{Z}$. The form of $f(x)$ satisfies the input of the rule RR10, then through applying the rule RR10 we obtain $f(x) = a_1 x \circ log_{a_2} x$. Now, the form of $f(x)$ satisfies the input of the rule RR5, then by applying the rule RR5 $f(x)$ can be rewritten as $f(x) = log_{a_1} x$. Consequently, we obtain $\sum f(x_i)$ is a natural instance of the symbolic aggregation state $\sum log_{p_1} x_i$ with a symbolic parameter p_1.*

The design principle of rewriting rules is reducing the length $|f|$ to $|f| - 1$, such that rules for the case of $|f| - 1$ can be reused. If scalar functions with a same length are equivalent, i.e. $a_3 x \circ log_{a_2} x \circ x^{a_1} = a_3 x \circ a_1 x \circ log_{a_2} x$ with constants a_3, a_2 and a_1 where $a_1 \neq 2k, k \in \mathbb{Z}$, they will be rewritten to one scalar function, i.e. $a_3 x \circ log_{a_2} x \circ x^{a_1} \rightarrow a_3 x \circ a_1 x \circ log_{a_2} x$, and we only need to maintain one rule to reduce the length, i.e. $a_3 x \circ a_2 x \circ log_{a_1} x \rightarrow a_3 a_1 x \circ log_{a_1} x$.

4.2 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. For example, $\sum px_i$ *shares* $\prod p^{x_i}$, since $\sum 5x_i$ shares $\prod 3^{x_i}$ and if we change 5 to be any other constant, e.g., 6, we still have $\sum 6x_i$ shares $\prod 3^{x_i}$. We formally define below such a sharing relationship as a symbolic derivable set of symbolic aggregation states.

Symbolic derivable set. Let ss be a symbolic aggregation state. The symbolic derivable set of ss, noted $SD(ss)$, is formally defined as follows:

$$SD(ss) = \{ss' | \text{for any natural instance } s' \text{ of } ss', \text{there exists a natural instance } s \text{ of } ss, \text{ such that } s' \in D(s)\}.$$

In the sequel, we say ss' shares ss when $ss' \in SD(ss)$.

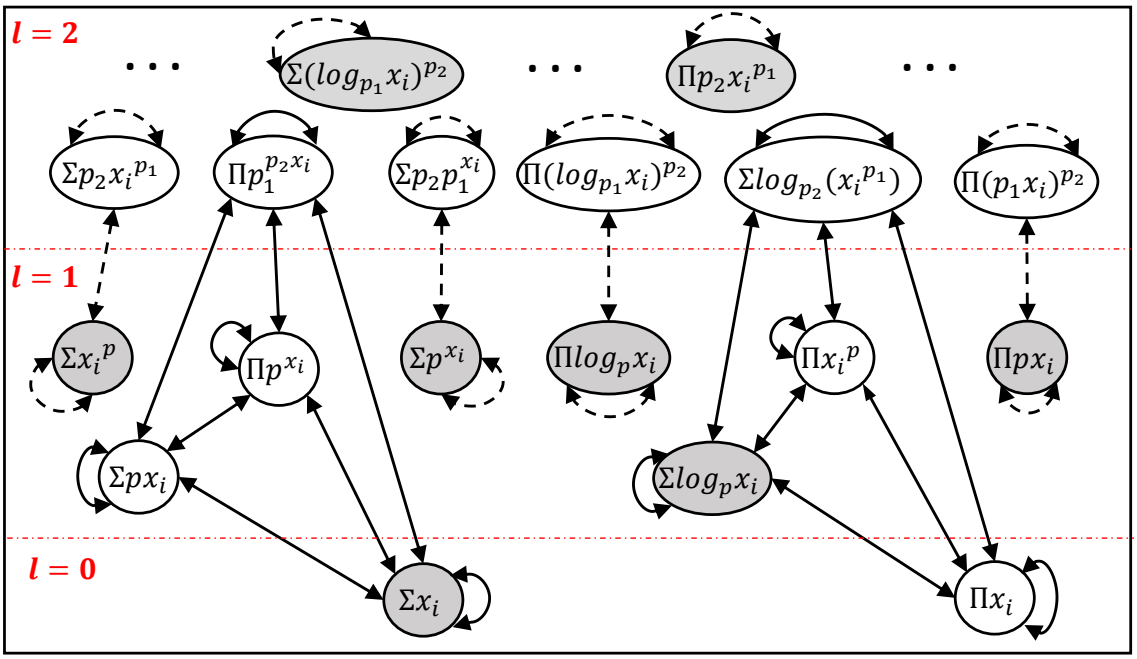

FIGURE 4.1: The digraph G of $saggs_2(X)$.

As explained previously, our aim is to precompute and store the sharing relationships between the 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 4.5, it was enough to use aggregation states, whose scalar functions have a length of up to 2 to express many useful and complex aggregation functions used in real-world applications. Therefore, SUDAF enables a user to bound the space of symbolic aggregation 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 l-bounded 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 $\frac{2(4^{l+1}-1)}{3}$.

Hence, once the parameter l is fixed by the user, SUDAF builds the 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 4.1, where each symbolic aggregation state is depicted as a node labeled with its expression (we shall explain later the meaning of edges between nodes in Figure 4.1). As it can be observed in Figure 4.1, 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 4.1 shows all the symbolic states of level 0 and 1, and some states of level 2.

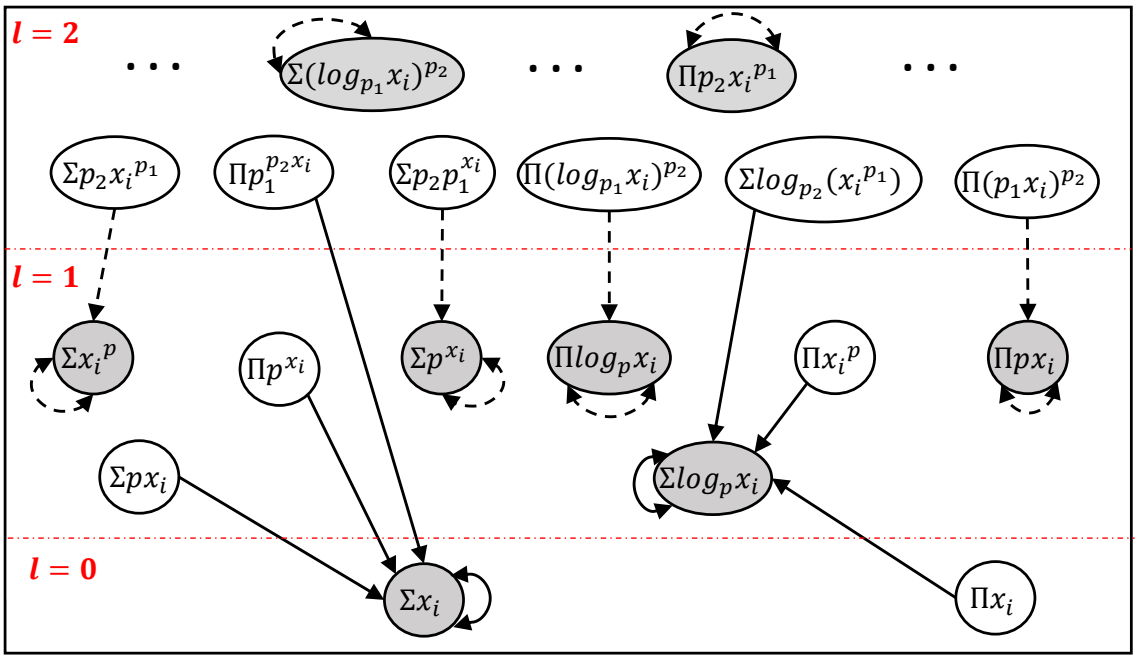

FIGURE 4.2: The simplified digraph G of $saggs_2(X)$.

4.3 Organizing the space $saggs_l(X)$

We briefly discuss the organization of a space $saggs_l(X)$, 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 aggregation 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. We shall present how to construct a digraph G in Section 4.3.1. Figure 4.1 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 4.1).

- *Strong relationships*. The first type of sharing relationships, called *strong relationships*, relate two symbolic states (ss', ss) if ss' shares ss without requiring any condition on the parameters. For example, since any instance of $\sum px_i$ shares any instance of $\prod p^{x_i}$ without conditions on their parameters, then $\sum px_i$ and $\prod p^{x_i}$ have a strong sharing relationship denoted as $\sum px_i \rightarrow \prod p^{x_i}$.

- *Weak relationships*. The second type of relationships, called *weak relationships*, relate two symbolic states (ss', ss) if ss' shares ss under some conditions defined over the parameters of ss and ss'. For 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 $\sum x_i^p \xrightarrow{p=p_1} \sum p_2 x^{p_1}$.

Figure 4.1 depicts the $saggs_2(X)$ space, with the exhaustive states of level 0 and 1, and an excerpt of states of level 2. The strongly shares relationships are depicted as directed solid-line edges while the weakly shares relationships are depicted using directed dashed-line edges. Interestingly, the following lemma states that the *shares* relationships is an equivalence relation (Proofs of Lemma 2 is provided in Section 4.3.2).

Lemma 2. *Let X be a multiset of positive values and let l be a positive integer. The shares relationship on the set $saggs_l(X)$ is an equivalence relation.*

For example, $\sum px_i \leftrightarrow \prod p^{x_i}$ and $\sum x_i^p \overset{p=p_1}{\leftrightarrow} \sum p_2 x^{p_1}$.

Consequently, the space $saggs_2(X)$ can be partitioned into *equivalence classes*. We define the equivalent class $[ss]$ associated with a symbolic state $ss \in saggs_l(X)$ as follows: $[ss] = \{ss' \in saggs_l(X) | ss' \in SD(ss)\}$. Intuitively, for a symbolic state ss, its associated equivalence class $[ss]$ is made of the set of symbolic aggregation states that share (and are shared by) ss. For example, as depicted in Figure 4.1:

$$[\sum x_i] = \{\sum x_i, \sum px_i, \prod p^{x_i}, \prod p_1^{p_2 x_i}\};$$
$$[\sum x_i^p] = \{\sum x_i^p, \sum p_2 x_i^{p_1}\}.$$

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 $[ss]$. Therefore, we propose the notion of a *representative* of an equivalence class $[ss]$, noted $rep([ss])$, which corresponds to a unique element in $[ss]$ whose instances are materialized in the cache. In our implementation, we use a hash function that maps symbolic states to integers. A representative $rep([ss])$ corresponds to the symbolic state of $[ss]$ having the smallest hash code. The hash function is designed to ensure the following three properties of representatives: a representative of an equivalence class is unique, representatives are selected among symbolic states with the smallest lengths and the priority is given to addition w.r.t. product in terms of binary operations $\oplus$.

We simplify the digraph G presented in Figure 4.2 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 $ss \overset{pcon}{\rightarrow} rep([ss])$ with a parameter condition ($pcon$). Consequently, when an instance s of ss is given, we use an edge $ss \rightarrow rep([ss])$, or $ss \overset{pcon}{\rightarrow} rep([ss])$ to get a cached instance of $rep([ss])$ to compute s.

Extension to an arbitrary multiset. When a multiset X contains negative values, the arisen issue is that some symbolic states in $saggs_2(X)$ do not exist, e.g., $\sum log_p x_i$. We deal with this issue by reducing this case to the case where an input contains only positive values. The main idea is to translate an input $X = \{x_1, \ldots, x_n\}$ to a multiset $\hat{X} = \{(|x_1|, sgn(x_1)), \ldots, (|x_n|, sgn(x_n)\}$, where $|x_j|$ denotes the absolute value of x_j and $sgn(x_j)$ is its sign. For example, for an arbitrary multiset X, we can keep in the cache states of the form $(\sum ln|x_i|, \prod sgn(x_i))$ corresponding to the representative $\sum log_p x_i$. By this way, it is possible to identify that an instance of $\log_{p_2}(x_i^{p_1})$ can be computed from the instance of $\sum log_p x_i$.

4.3.1 Construction of graph G

In this section, we present an approach of constructing a sharing graph G associated with a space $saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$. Constructing a graph G requires to check the property of symbolic derivable set for every pair of two symbolic aggregation states (ss', ss) in $saggs_l(X) \times saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$. The key problem is to check whether $ss' \in SD(ss)$. In order to accomplish this goal, we reuse the sharing conditions in proposition 1 (a partial result of Theorem 3) to verify $ss' \in SD_l(ss)$. We propose complete conditions to analyze the forms of symbolic scalar functions, such that to identify strong edges, i.e. when ss' strongly shares ss. We also propose a sound approach to construct weak edges, i.e. when ss' weakly shares ss. At last, we analyze the effect of using a constructed graph

G to search a natural instance for computing an aggregation state which is an unnatural instance of a symbolic state.

Symbolic inverse functions. Verifying sharing conditions require to obtain an inverse function of a scalar function. We define symbolic inverse functions of symbolic scalar functions as follows. We first assign a symbolic inverse function for each symbolic primitive scalar function in Table 4.1.

$sf_{\bar{p}}(x)$	$sf_{\bar{p}}^{-1}(x)$
px	$1/px$
$log_p x$	p^x
p^x	$log_p x$
x^p	$x^{1/p}$

TABLE 4.1: Symbolic inverse $sf_{\bar{p}}^{-1}(x)$ of symbolic primitive scalar function $sf_{\bar{p}}(x), x > 0$.

Then, given a symbolic scalar function $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x), x > 0$, where $sf_{i\bar{p}_i}(x)$ is a symbolic primitive scalar function, $i \in [1, ..., l]$, we define the symbolic function $sf_{\bar{p}}^{-1}(x) = sf_{1\bar{p}_1}^{-1} \circ ... sf_{l\bar{p}_l}^{-1}(x)$ as the symbolic inverse function of $sf_{\bar{p}}(x)$, where $sf_{i\bar{p}_i}^{-1}(x)$ is the symbolic inverse of $sf_{i\bar{p}_i}(x)$.

Symbolic sharing conditions. We transform the sharing conditions of concrete aggregation states (proposition 1) to the version on symbolic aggregation states, which can be used in the symbolic context for the case $X \in \mathcal{M}(\mathbb{Q}_+)$. We make a decision according to $sf'_{\bar{p}'} \circ sf_{\bar{p}}^{-1}(x)$ based on $\sum_{\oplus'}$ and $\sum_{\oplus}$. If $sf'_{\bar{p}'} \circ sf_{\bar{p}}^{-1}(x)$ is equivalent to a specific one in the set $\{px, p'(log_p x), p^{p''x}, x^p\}$ according to $\sum_{\oplus'}$ and $\sum_{\oplus}$ with symbolic parameters p, p' and p'', then we have $ss' \in SD_l(ss)$ and (ss', ss) is a strong edge.

Identifying strong edges. Let $sf'_{\bar{p}'} \circ sf_{\bar{p}}^{-1}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x), l \geqslant 2$ and $sf_{i\bar{p}_i}, i \in [1, ..., l]$ be a symbolic primitive scalar function, we present below Lemma 3 to identify whether $sf'_{\bar{p}'} \circ sf_{\bar{p}}^{-1}(x)$ is one in the set $\{px, p'(log_p x), p^{p''x}, x^p\}$ with symbolic parameters p, p' and p''. The proof of Lemma 3 is based on induction, and the details of the proof are provided in Section 4.3.3.

Lemma 3. *Given a symbolic scalar function $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x)$ where $sf_{i\bar{p}_i}(x), i \in [1, ..., l], l \geqslant 2, x > 0$ is a symbolic primitive scalar function with symbolic parameter p_i, then we have the following conditions to identify when $sf_{\bar{p}(x)}$ is one in the set $\{px, x^p, p'log_p x, p^{p''x}\}$ with symbolic parameters p, p' and p',*

- ***(Case 1)*** *$sf_{\bar{p}}(x) = px$, if and only if*
  - $\forall sf_{i\bar{p}_i}(x) = p_i x$;
  - *or* $\forall sf_{i\bar{p}_i}(x) \neq p_i x, sf_{i\bar{p}_i}(x) = log_{p_i} x, sf_{j\bar{p}_j}(x) = {p_j}^x, j = i-1, i > 1$;
  - *or* $\forall sf_{i\bar{p}_i}(x) \neq p_i x, sf_{i\bar{p}_i}(x) = {p_i}^x, sf_{j\bar{p}_j}(x) = log_{p_j} x, j = i+1, i < l$.
- ***(Case 2)*** *$sf_{\bar{p}}(x) = x^p$, if and only if*
  - $\forall sf_{i\bar{p}_i}(x) = x^{p_i}$;
  - *or* $\forall sf_{i\bar{p}_i}(x) \neq x^{p_i}, sf_{i\bar{p}_i}(x) = {p_i}^x, sf_{j\bar{p}_j}(x) = log_{p_j} x, j = i-1, i > 1$;
  - *or* $\forall sf_{i\bar{p}_i}(x) \neq x^{p_i}, sf_{i\bar{p}_i}(x) = log_{p_i} x, sf_{j\bar{p}_j}(x) = {p_j}^x, j = i+1, i < l$.
- ***(Case 3)*** *$sf_{\bar{p}}(x) = p'log_p x$, if and only if*

  - $sf_{i\bar{p}_i}(x) = log_{p_i}x$, $sf_{l\bar{p}_l} \circ ... \circ sf_{i+1\bar{p}_{i+1}}(x) = p'x$, *and* $sf_{l\bar{p}_{i-1}} \circ ... \circ sf_{1\bar{p}_1}(x) = x^{p''}$, $l > i > 1$.
  - *or* $sf_{l\bar{p}_l}(x) = log_{p_l}x$, *and* $sf_{l-1\bar{p}_{l-1}} \circ ... \circ sf_{1\bar{p}_1}(x) = x^{p''}$.
  - *or* $sf_{1\bar{p}_1}(x) = log_{p_1}x$, *and* $sf_{l\bar{p}_l} \circ ... \circ sf_{2\bar{p}_2}(x) = p'x$.

- ***(Case 4)*** $sf_{\bar{p}}(x) = p^{p''x}$, *if and only if*

  - $sf_{i\bar{p}_i}(x) = p_i^x$, $sf_{l\bar{p}_l} \circ ... \circ sf_{i+1\bar{p}_{i+1}}(x) = x^{p'}$, *and* $sf_{l\bar{p}_{i-1}} \circ ... \circ sf_{1\bar{p}_1}(x) = p''x$, $l > i > 1$.
  - *or* $sf_{l\bar{p}_l}(x) = p_l^x$, *and* $sf_{l-1\bar{p}_{l-1}} \circ ... \circ sf_{1\bar{p}_1}(x) = p''x$.
  - *or* $sf_{1\bar{p}_1}(x) = p_1^x$, *and* $sf_{l\bar{p}_l} \circ ... \circ sf_{2\bar{p}_2}(x) = x^{p'}$.

We identify a property [1] of strong relationships by using Lemma 3: *if* (ss', ss) *is a strong edge, then we have either* $ss' \in SD_l(\sum x_i)$ *or* $ss' \in SD_l(\prod x_i)$. Moreover, since sharing relationship is equivalent, then we can also have $ss \in SD_l(\sum x_i)$ if $ss' \in SD_l(\sum x_i)$, or $ss \in SD_l(\prod x_i)$ if $ss' \in SD_l(\prod x_i)$. This property provides a way to identify all strong edges in $saggs_l(X)$ without verifying all pairs from $saggs_l(X) \times saggs_l(X)$. Specifically, we only need to take every symbolic state ss' from $saggs_l(X)$ and check whether $ss' \in SD_l(\sum x_i)$ or $SD_l(\prod x_i)$. This property also states that there only exist two such equivalent classes where symbolic aggregate states strongly share each other. Once all strong edges are constructed in G, $\sum x_i$ and $\sum log_p x_i$ are selected to be the representative in each of the two equivalent classes.

Identification of weak edges. We propose an incremental approach to obtain weak edges. We first group symbolic aggregation states $ss(X) = \sum_{\oplus} sf_{\bar{p}}(x_i)$ based on $|sf_{\bar{p}}|$, and we call each group a layer, i.e. the layer l contains all symbolic aggregation states with $|sf_{\bar{p}}| = l$. In the following context, we use the notion *weak representative* to denote a representative that weakly shares itself. We begin to identify weak edges and weak representatives in the layer 1. Then, we construct weak edges from symbolic aggregation states in the layer $i, i \geq 2$ to the weak representatives in the layer $i-1$, and we identify weak representatives in the layer i. We explain the incremental approach as follows.

- (Layer 1). As illustrated in Figure 4.1, there exists 4 symbolic aggregation states, $\sum x_i^p$, $\sum p^{x_i}$, $\prod px_i$ and $\prod log_p x_i$, which have the following property by using sharing conditions in proposition 1: instances of them do not share each other. Therefore, we construct for each a weak edge pointing to itself, and we maintain for each weak edge a condition, that an instance only shares itself. These 4 symbolic states are identified as 4 weak representatives. We shall discuss that they will be reused to constructed weak edges for elements in the layer 2.

  Example 8. $\sum x_i^p$ *is identified as a weak representative in the layer* 1, *and we construct a weak edge* $(\sum x_i^p, \sum x_i^p)$. *We also maintain the following condition associated with the weak edge: given instances of* $\sum x_i^p$, *they can share computations unless they have the identical constant for the symbolic parameter* p. *For example,* $\sum x_i^2$ *does not share* $\sum x_i^3$, *and* $\sum x_i^2$ *only shares* $\sum x_i^2$. *Note that, if an unnatural instance of* $\sum x_i^p$ *is given, i.e.* $\sum x_i$, *the unnatural instance will not share any instance besides itself.*

- (Layer 2 or higher). Given a symbolic aggregation state ss_2 in the layer 2, and let $s_2(X) = \sum_{\oplus} f_2 \circ f_1(x_i)$ be a natural instance of ss_2, we have: if $\sum_{\oplus} f_2(x_i)$ is one

[1] This can be verified by exhaustively checking the shape of the symbolic scalar function $sf'_{\bar{p}'}(x)$ of ss'.

in the set $\{\sum ax_i, \sum log_a x_i, \prod a^{x_i}, \prod x_i^a\}$ with a constant a, then $s_2(X) = f_2 \circ s_1(X)$, where $s_1(X) = \sum_{\oplus'} f_1(x_i)$ is a natural instance of a symbolic aggregation state ss_1 from the layer 1. In this case, if ss_1 is a weak representative, i.e. one in the set $\{\sum x_i^p, \sum p^{x_i}, \prod px_i, \prod log_p x_i\}$, then we construct a weak edge (ss_2, ss_1) with the condition of identical $f_1(x)$.

Example 9. *For an instance $\sum a_2 x_i^{a_1}$ of the symbolic aggregation state $\sum p_2 x \circ x_i^{p_1}$, we have $\sum a_2 x_i^{a_1} = a_2(\sum x_i^{a_1})$, and $\sum x_i^{a_1}$ is an instance of the weak representative $\sum x_i^p$. Such that, we construct a weak edge $(\sum p_2 x \circ x_i^{p_1}, \sum x_i^p)$ with the condition $p_1 = p$.*

In the layer 2, we also meet such symbolic states: they do not weakly share any element from the layer 1, and they do not strongly share any element in a space $saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$. For each of such elements, we let it be a weak representative, and we construct a weak edge pointing to itself with the condition, that an instance of such symbolic state only shares itself. Then, we terminate for the layer 2.

Example 10. *$\sum x^{p_2} \circ log_{p_1} x_i$ does not weakly share any element from the layer 1, and it does not strongly share any element in $saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$. Then, it is identified as a weak representative, and we construct $(\sum x^{p_2} \circ log_{p_1} x_i, \sum x^{p_2} \circ log_{p_1} x_i)$ with the condition, that an instance of $\sum x^{p_2} \circ log_{p_1} x_i$ only shares itself.*

We repeat the above steps to construct weak edges from symbolic aggregation states in the layer 3 to weak representatives in the layer 2, and we identify weak representatives in the layer 3. We incrementally construct weak edges and identify weak representatives until to the layer l, such that to obtain weak edges and weak representatives in a graph G associated with a symbolic space $saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$.

Complexity and completeness. We separately only need to visit each element in $saggs_l(X)$ at most twice to construct strong edges and weak edges, such that a sharing graph G is constructed in $O(|saggs_l(X)|)$ time, which is $O(4^l)$ since $|saggs_l(X)|$ is bounded by $\frac{2(4^{l+1}-1)}{3}$. The graph G of a space $saggs_l(X), X \in \mathcal{M}(\mathbb{Q}_+)$ with a general l is partially complete, because strong edges have been exhaustively identified, but the identification of weak edges maybe not complete. As a consequence, there may exist aggregation states, which are natural instances of different weak representatives (representatives weakly share itself) and can share each other, and such sharing relationship is not identified by the graph G.

Sharing unnatural instances using edges in G. We analyze what are the consequences of using edges in G to search a natural instance of a symbolic state for computing an unnatural instance of a symbolic state. Using strong edges can still share computation and ensure no redundancies in caches since strong edges do not require a condition to share a natural instance of a representative. While, weak edges contain a condition to specify which instance of a representative can be shared, such that using weak edges can share computation, but an unnatural instance of a symbolic state will be cached which creates redundancies. For example, if we do not maintain reduction rules, given $\sum x_i^a, a = 1$, that is an unnatural instance of $\sum x_i^p$, it will be cached under the representative $\sum x_i^p$, and it cannot be shared by other symbolic aggregations states associated with the representative $\sum x_i$, i.e. $\sum px_i$.

4.3.2 Proof of Lemma 2

In this section, we provide a proof of Lemma 2. We first recall sharing conditions in the case 2 of Theorem 3 and propose Lemma 4, based on which we prove Lemma 2.

We refine the sharing conditions in Theorem 3 and propose proposition 1, which can be seen as partial sharing conditions in the sense that X only contains positive values.

Proposition 1. *Let $X \in \mathcal{M}(\mathbb{Q}_+)$, and $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be two aggregation states in aggs(X). Then, we have: $s_1 \in D(s_2)$ 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.*

(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.*

(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.*

(2.4) $\sum_{\oplus_1} = \sum_{\oplus_2} = \prod$ *and* $f_1 \circ f_2^{-1}(x) = x^a$ *with a constant* $a \in \mathbb{Q}_{\neq 0}$.

Proof. When $X \in \mathcal{M}(\mathbb{Q}_+)$, we only need to consider the scalar function $f_1(x)$ and $f_2(x)$ over the domain $\mathbb{Q}_+$. We can derive that both $f_1(x)$ and $f_2(x)$ are injections over the domain $\mathbb{Q}_+$. Let $f_1 \circ f_2^{-1}(x) = f_3(x)$, then we have $f_1(x) = f_3 \circ f_2(x)$. Assuming $f_1 : \mathbb{Q}_+ \to D_1$ and $f_2 : \mathbb{Q}_+ \to D_2$, then we have $f_3 : D_2 \to D_1$ is an injection, because in our context f_1 and f_2 are constructed by composing primitive scalar functions, and every primitive scalar function is an injections and bijection over the domain $\mathbb{Q}_+$. Such that, we have $f_3(x)$ is also an injection. Then according to the case 2 of Theorem 3, we obtain the forms of $f_3(x)$ when $f_3(x)$ is an injection. □

Lemma 4. *Let ss_1 and ss_2 be two symbolic aggregation states from $saggs_l(X)$, $X \in \mathcal{M}(\mathbb{Q}_+)$, if there exists a natural instance s_1 of ss_1 shares a natural instance s_2 of ss_2, then $ss_1 \in' (ss_2)$.*

Proof. We prove that, if there exists a natural instance of ss_1 shares a natural instance of ss_2, we can obtain that for any natural instance of ss_1, there exists a natural instance ss_2 that can be shared.

Let $ss_1(X) = \sum_{\oplus_1} sf_{1\bar{p}_1}(x_i)$ and $ss_2(X) = \sum_{\oplus_2} sf_{2\bar{p}_2}(x_i)$ be two symbolic aggregation states. Let $s_1(X) = \sum_{\oplus_1} f_1(x_i)$ and $s_2(X) = \sum_{\oplus_2} f_2(x_i)$ be separately their natural instances. Assuming s_1 shares s_2, and let $f_1 \circ f_2^{-1}(x) = f_3(x)$, then according to the case 2 of proposition 1, we have:

(i) if $\sum_{\oplus_1} = \sum_{\oplus_2} = \sum$, then $f_3(x) = ax$ with a constant a.

(ii) if $\sum_{\oplus_1} = \sum$ and $\sum_{\oplus_2} = \prod$, then $f_3(x) = a(log_b x)$ with constants a and b.

(iii) if $\sum_{\oplus_1} = \prod$ and $\sum_{\oplus_2} = \sum$, then $f_3(x) = b^{ax}$ with constants a and b.

(vi) if $\sum_{\oplus_1} = \sum_{\oplus_2} = \prod$, then $f_3(x) = x^a$ with a constant a.

We obtain that $f_3(x)$ has the following property $\sum_{\oplus_1} f_1(x_i) = \sum_{\oplus_1} f_3 \circ f_2(x_i) = f_3(\sum_{\oplus_2} f_2(x_i))$. For instance, if $\sum_{\oplus_1} = \sum_{\oplus_2} = \sum$, then $f_3(x) = ax$ with a constant a, then $\sum f_1(x_i) = a(\sum f_2(x_i))$.

Knowing that an aggregation state is a natural instance of a unique symbolic aggregation state. Since $s_1(X)$ is a natural instance of $ss_1(X)$, and $s_1(X) = \sum_{\oplus_1} f_3 \circ f_2(x_i)$, such that we have $ss_1(X) = \sum_{\oplus_1} sf_{3\bar{p}_3} \circ sf_{2\bar{p}_2}(x_i)$, where $f_3(x)$ is an instance of $sf_{3\bar{p}_3}(x)$ and $f_2(x)$ is an instance of $sf_{2\bar{p}_2}(x)$.

	p_2x	x^{p_2}	$log_{p_2}x$	${p_2}^x$
p_1x	$p_2x \circ p_1x = p_2p_1x$	$x^{p_2} \circ p_1x$	$log_{p_2}x \circ p_1x$	${p_2}^x \circ p_1x = ({p_2}^{p_1})^x$
x^{p_1}	$p_2x \circ x^{p_1}$	$x^{p_2} \circ x^{p_1} = x^{p_2p_1}$	$log_{p_2}x \circ x^{p_1} = p_1x \circ log_{p_2}x$	$p_2^x \circ x^{p_1}$
$log_{p_1}x$	$p_2x \circ log_{p_1}x = log_{p_1}x \circ x^{p_2}$	$x^{p_2} \circ log_{p_1}x$	$log_{p_2}x \circ log_{p_1}x$	${p_2}^x \circ log_{p_1}x = x^{1/log_{p_2}p_1}$
${p_1}^x$	$p_2x \circ {p_1}^x$	$x^{p_2} \circ {p_1}^x = (p_1^{p_2})^x$	$log_{p_2}x \circ {p_1}^x = (log_{p_2}p_1)x$	${p_2}^x \circ {p_1}^x$

TABLE 4.2: Compositions of symbolic primitive scalar functions $sf_{2\bar{p}_2} \circ sf_{1\bar{p}_1}(x), x > 0$.

Let $\sum_{\oplus_1} f_1'(x_i)$ with $f_1'(x) = f_3' \circ f_2'(x)$ be any natural instances of ss_1, where $f_3'(x)$ is an instance of $sf_{3\bar{p}_3}(x)$ and $f_2'(x)$ is an instance of $sf_{2\bar{p}_2}(x)$. Note that, $\sum_{\oplus_2} f_2'(x_i)$ is also a natural instance of ss_2, otherwise $\sum_{\oplus_1} f_1'(x_i)$ can not be a natural instance of ss_1. Then, since both $f_3(x)$ and $f_3'(x)$ are instance of $sf_{3\bar{p}_3}(x)$, we can also obtain $\sum_{\oplus_1} f_1'(x_i) = \sum_{\oplus_1} f_3' \circ f_2'(x_i) = f_3'(\sum_{\oplus_2} f_2'(x_i))$. For instance, if $\sum_{\oplus_1} = \sum_{\oplus_2} = \sum$, then $f'(x) = a'x$ with a constant a', then $\sum f_1'(x_i) = a'(\sum f_2'(x_i))$.

Consequently, for any natural instance $\sum_{\oplus_1} f_1'(x_i)$ of ss_1, there exists a natural instance $\sum_{\oplus_2} f_2'(x_i)$ of ss_2, such that $\sum_{\oplus_1} f_1'(x_i)$ shares $\sum_{\oplus_2} f_2'(x_i)$. Therefore, $ss_1 \in SD(ss_2)$. □

4.3.3 Proof of Lemma 3

In this section, we provide the proof of Lemma 3 as follows. The proof is based on induction, and the initial cases are presented in Table 4.2.

Proof. **(Case 1 of Lemma 3)**

(Sufficiency) Straightforward.

(Necessity) We prove this case by induction. We first prove it is true for the initial case $l = 2$. Then assuming it is true for l, we prove it is also true for $l + 1$.

The initial case is $l = 2$, that $sf_{\bar{p}}(x) = sf_{2\bar{p}_2} \circ sf_{1\bar{p}_1}(x) = px$. We present all compositions of two symbolic primitive scalar functions in Table 4.2. We observe that, $sf_{\bar{p}}(x) = px$ if and only if $sf_{2\bar{p}_2}(x) = p_2x$ and $sf_{1\bar{p}_1}(x) = p_1x$, or $sf_{2\bar{p}_2}(x) = log_{p_2}x$ and $sf_{1\bar{p}_1}(x) = p_1^x$. Then, it is true for this case.

We assume it is true for $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x) = px$, then we prove it is also true for the case $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = p''x$, where $sf'_{1\bar{p}'_1}(x)$ is a symbolic primitive scalar function. Since, $sf_{\bar{p}}(x) = px$, such that we have $px \circ sf'_{1\bar{p}'_1}(x) = p''x$. Then, according to Table 4.2, we have that $sf'_{1\bar{p}'_1}(x)$ can only be a symbolic linear scalar function, that $sf'_{1\bar{p}'_1}(x) = p'x$. Therefore, $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = p''x$ still has the property. □

Proof. **(Case 2 of Lemma 3)** The proof follows the same structure as the one of case 1.

(Sufficiency) Straightforward.

(Necessity) We also prove this case by induction.

The initial case is $l = 2$, that $sf_{\bar{p}}(x) = sf_{2\bar{p}_2} \circ sf_{1\bar{p}_1}(x) = x^p$. We present all compositions of two symbolic primitive scalar functions in Table 4.2. We observe that, $sf_{\bar{p}}(x) = x^p$ if and only if $sf_{2\bar{p}_2}(x) = x^{p_2}$ and $sf_{1\bar{p}_1}(x) = x^{p_1}$, or $sf_{2\bar{p}_2}(x) = p_2^x$ and $sf_{1\bar{p}_1}(x) = log_{p_1}x$. Then, it is true for this case.

We assume it is true for $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x) = x^p$, then we prove it is also true for the case $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = x^{p''}$, where $sf'_{1\bar{p}'_1}(x)$ is a symbolic primitive scalar function. Since, $sf_{\bar{p}}(x) = x^p$, such that we have $x^p \circ sf'_{1\bar{p}'_1}(x) = x^{p''}$. Then, we have $sf'_{1\bar{p}'_1}(x)$ can only be a symbolic power scalar function, that $sf'_{1\bar{p}'_1}(x) = x^{p'}$. Therefore, $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = x^{p''}$ still has the property. □

Proof. **(Case 3 of Lemma 3)** The proof follows the same structure as the one of cases 1 and 2.

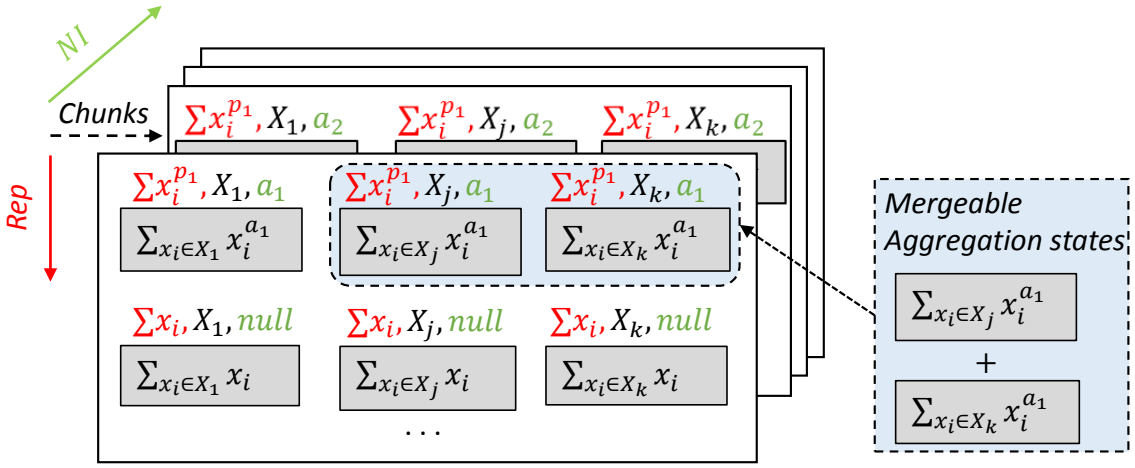

FIGURE 4.3: SUDAF cache.

(Sufficiency) Straightforward.

(Necessity) We also prove this case by induction.

The initial case is $l = 2$, that $sf_{\bar{p}}(x) = sf_{2\bar{p}_2} \circ sf_{1\bar{p}_1}(x) = p' log_p x$. We present all compositions of two symbolic primitive scalar functions in Table 4.2. We observe that, $sf_{\bar{p}}(x) = p' log_p x$ if and only if $sf_{2\bar{p}_2}(x) = p_2 x$ and $sf_{1\bar{p}_1}(x) = log_{p_1} x$, or $sf_{2\bar{p}_2}(x) = log_{p_2} x$ and $sf_{1\bar{p}_1}(x) = x^{p_1}$. Then, it is true for this case.

We assume it is true for $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x) = p' log_p x$, then we prove it is also true for the case $sf_{\bar{p}}(x) \circ sf''_{1\bar{p}''_1}(x) = p''' log_p x$, where $sf''_{1\bar{p}''_1}(x)$ is a symbolic primitive scalar function. Since $sf_{\bar{p}}(x) = p' log_p x$, such that we have $p' log_p x \circ sf''_{1\bar{p}''_1}(x) = p''' log_p x$. Then, we have $sf''_{1\bar{p}''_1}(x)$ can only be a symbolic power scalar function, that $sf''_{1\bar{p}''_1}(x) = x^{p''}$. Therefore, $sf_{\bar{p}}(x) \circ sf''_{1\bar{p}''_1}(x) = p''' log_p x$ still has the property. □

Proof. **(Case 4 of Lemma 3)** The proof follows the same structure as the one of cases 1, 2 and 3.

(Sufficiency) Straightforward.

(Necessity) We also prove this case by induction.

The initial case is $l = 2$, that $sf_{\bar{p}}(x) = sf_{2\bar{p}_2} \circ sf_{1\bar{p}_1}(x) = p^{p'' x}$. We present all compositions of two symbolic primitive scalar functions in Table 4.2. We observe that, $sf_{\bar{p}}(x) = p^{p'' x}$ if and only if $sf_{2\bar{p}_2}(x) = p_2^x$ and $sf_{1\bar{p}_1}(x) = p_1 x$, or $sf_{2\bar{p}_2}(x) = x^{p_2}$ and $sf_{1\bar{p}_1}(x) = p_1^x$. Then, it is true for this case.

We assume it is true for $sf_{\bar{p}}(x) = sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x) = p^{p'' x}$, then we prove it is also true for the case $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = p^{p''' x}$, where $sf'_{1\bar{p}'_1}(x)$ is a symbolic primitive scalar function. Since $sf_{\bar{p}}(x) = p^{p'' x}$, such that we have $p^{p'' x} \circ sf'_{1\bar{p}'_1}(x) = p^{p''' x}$. Then, according to Table 4.2, we can have that $sf'_{1\bar{p}'_1}(x)$ can only be a symbolic linear scalar function, that $sf'_{1\bar{p}'_1}(x) = p' x$. Therefore, $sf_{\bar{p}}(x) \circ sf'_{1\bar{p}'_1}(x) = p^{p''' x}$ still has the property. □

4.4 Anatomy of the SUDAF cache

We describe in this section the structure of the SUDAF cache and its associated index. As depicted in Figure 4.3, conceptually the SUDAF cache can be seen as a 3D structure $(Rep, Chunks, NI)$, where Rep is the dimension of the representatives of the equivalent classes of the space $saggs_l(X)$, $Chunks$ is a data partition dimension, i.e., $X_1, \ldots, X_k$,

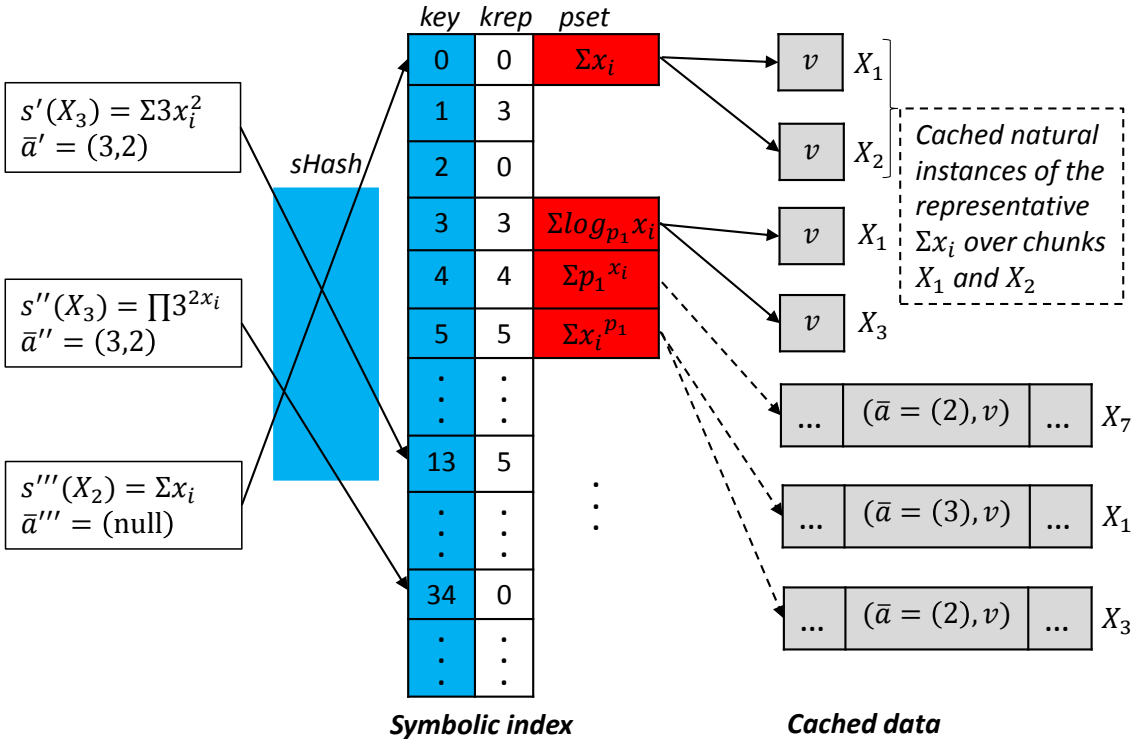

FIGURE 4.4: Implementation of SUDAF cache.

where each X_i is a chunk, and NI is the dimension of natural instances actually stored in the cache. Figure 4.4 shows the implementation of the SUDAF cache. The cache includes a symbolic index that contains pointers to cached data. More precisely, an entry in a symbolic index associates a symbolic state $ss(X) = \sum_{\oplus} sf_{\bar{p}}(x_i)$ of the space $saggs_l(X)$ with a representative $rep([ss])$ of an equivalent class $[ss]$. Each representative is also associated with a set of pointers to lists of cached natural instances. Each list of cached data corresponds to a given chunk X_i and is made of pairs $(\bar{a}, v)$ where $\bar{a}$ is a sequence of constants and v is the result of the natural instance $\sum_{\oplus} sf_{\bar{a}}(x_i)$ of the symbolic state $ss(X)$. In addition, we implemented a hash function, denoted $sHash$, that enables to associate a natural instance to an entry in the symbolic index. The following example illustrates the use of the symbolic index.

Example 11. *Let $s'(X_3) = \sum 3x_i^2$ be a new aggregation state to compute. First, SUDAF computes the symbolic state $ss' = \sum p_2' x_i^{p_1'}$ such that s' is a natural instance of ss' and then applies the sHash function to generate the symbolic code of ss'. As shown in the Figure 4.4, its symbolic hash code is 13 (i.e., $sHash(ss') = 13$). The symbolic code enables to identify the representative $rep([ss'])$ of the equivalence class $[ss']$ using the symbolic index. In our example, the key $sHash(ss') = 13$ in the symbolic index matches with the identifier 5, which is the key of the representative $rep([ss']) = \sum x_i^{p_1}$ in the symbolic index. Then, using the entry 5 of the symbolic index, SUDAF uses an associated pointer set to get the pointer corresponding to the chunk X_3 in order to get the list of cached natural instances of the symbolic state $rep([ss'])$. Note that, this pointer is depicted in Figure 4.4 using a dot line because it corresponds to a weak sharing relationship between $ss' = \sum p_2' x_i^{p_1'}$ and $rep([ss']) = \sum x_i^{p_1}$ which means that the natural instances of ss' share the natural instances of $rep([ss'])$ under the condition that $p_1' = p_1$. Since in our example $p_1' = 2$, this translates to condition $\bar{a} = (2)$. To sum up, SUDAF uses the symbolic entry 13 to get a list of pairs $(\bar{a}, v)$ corresponding the the natural instances of the symbolic state $rep([ss'])$ over the chunk X_3 and then selects the pair $(\bar{a} = (2), v)$ to be reused to compute $s'(X_3) = \sum 3x_i^2$.*

Note that, SUDAF is able to compute the reuse function $p'_2 x$, with $p'_2 = 3$, which is needed to compute $\sum 3x_i^2$ from the cached state $\sum x_i^2$.

As another example, assume that $s''(X_3) = \prod 3^{2x_i}$ is a new aggregation state to compute. Following the same procedure, we obtain $sHash(ss'') = 34$, which is matched to the identifier 0 corresponding to the representative $rep([ss'']) = \sum x_i$ in the symbolic index. However, as it can be observed in Figure 4.4, there is no data cached for the chunk X_3 corresponding to the entry 0 in the symbolic index. In this case, a cache miss is detected and a new aggregation state $\sum x_i$ over the chunk X_3 is computed and stored in the cache. The new cached state is then used to compute the state $s''(X_3) = \prod 3^{2x_i}$. This example shows one interesting functionality of SUDAF: instead of caching the result of $s''(X_3) = \prod 3^{2x_i}$, SUDAF will compute and cache $\sum x_i$, the representative of its equivalence class.

Building the SUDAF cache. Let l be an integer. We explain below how the SUDAF cache of level l is constructed. We emphasis the fact that the construction of the cache corresponds to an initialization step of the SUDAF which is executed once when the system is installed. From the practical point of view, to build the SUDAF index, we first construct a graph $G = (V, E)$ of the space $saggs_l(X)$ with the set of nodes $V = saggs_l(X)$ and where the edges $E \subseteq V \times V$ represent the shares relationship, i.e., $(ss', ss) \in E$ iff $ss' \in SD_l(ss)$. The construction of the graph (described in Section 4.3.1) uses the sharing conditions provided in Theorem 3 extended to symbolic states. Note that, while building the graph G, we obtain for each edge $e = (ss', ss) \in E$ a symbolic reuse function $sr_{\bar{p}}$ which satisfies the following property: For any natural instance s of ss and any natural instance s' of ss', if s' shares s then there exists a constant sequence $\bar{a}$ such that $s'(X) = sr_{\bar{a}} \circ s(X)$. Moreover, in the case of a weak relationship e, we also compute and store the sharing conditions associated with e.

Symbolic index. The symbolic index is a 2-columns table $(key, krep)$, where the column key is used to store the identifiers of symbolic states of $saggs_l(X)$ and the column $krep$ is used to store the identifiers of representatives of equivalent classes of $saggs_l(X)$. For an entry $sHash(ss)$ in the symbolic index, if $sHash(ss)$ is identical to the identifier stored in the corresponding cell $krep$, then the entry is associated with a pointer set $pset$, which provides a set of pairs $(Chunk\_id, pt)$ each of which associate to a chunk X_i, identified by $Chunk\_id$, and a pointer pt to the list of natural instances of $rep([ss])$ actually stored in the cache. Hence, the entries of the symbolic index are given by the symbolic codes of the symbolic states ss in $saggs_l(X)$. The graph G is exploited to construct the symbolic index. Mapping ss to $rep([ss])$ is given by the edges of the graph G. We distinguish between two types of pointers to cached data: strong pointers (depicted by plain lines in Figure 4.4) that map strong sharing relationships and weak pointers (depicted with dotted lines in Figure 4.4) that map weak sharing relationships. A strong pointer leads to a unique natural instance while a weak pointer leads to a list of natural instances each of which identified by its constant sequence $\bar{a}$ and a sharing condition derived from the graph G.

The symbolic hash function $sHash$**.** SUDAF implements $sHash$, a function that maps a symbolic aggregation state $ss \in saggs_l(X)$ to an integer $sHash(ss)$ that uniquely identify ss. Given $ss(X) = \sum_{\oplus} sf_{l\bar{p}_l} \circ ... \circ sf_{1\bar{p}_1}(x_i)$, the $sHash$ function is defined as follows,

$$sHash(ss) = encode(sf_{l\bar{p}_l}) \times 4^{l-1} + \cdots + encode(sf_{1\bar{p}_1}) \times 4^0 + off(\sum_{\oplus}) + 2 \times (4^l - 4)/3 + 2$$

where $encode()$ is the following encoding function for symbolic primitive scalar function: $encode(px) = 0$, $encode(log_p x) = 1$, $encode(p^x) = 2$ and $encode(x^p) = 3$, with p a

parameter, and $off()$ is an offset function for primitive aggregate functions defined as follows: $off(\sum) = 0$ and $off(\prod) = 4^l$. For the two special elements $\sum x_i$ and $\prod x_i$, we let $sHash(\sum x_i) = 0$ and $sHash(\prod x_i) = 1$. For example, $sHash(\sum p_2 x_i^{p_1}) = 13$.

Complexity analysis. Let l be an integer. Building the symbolic index of the SUDAF cache can be done in time $O(4^l)$. Given an aggregation state $s(X_i) = \sum_{\oplus} f(x_i)$ over a chunk X_i with $|f| \leqslant l$, using the symbolic index of SUDAF to retrieve one of the m cached aggregation states to compute s takes $O(1)$ time when using strong pointers and takes $O(log(m))$ time when using weak pointers.

Data partitioning. Note that, the important issues related to selecting chunk size, data partitioning and identification of the chunks covered by a given query are not addressed in this paper. We refer the reader to existing techniques, e.g., proposed in [DRSN98, WWDI17], to deal with such issues. Caches of aggregation states on chunks can be trivially merged by exploiting the associative and commutative property of $\sum_{\oplus}, \sum_{\oplus} \in PA$.

4.5 Experimental evaluation

The general scheme of our experiments is the following. We select 3 query models, and we instantiate each query model using 11 aggregate functions. We simulate the 11 instances of each query model coming in 2 different orders, i.e., two different sequences of 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 or 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 functions 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. Their partial aggregation states are automatically rewritten using built-in functions by SUDAF, e.g., $\sum x_i^2$ is rewritten as sum(power(X,2)) and $\prod x_i$ is rewritten as exp(sum(ln(abs(X)))). And in case (iii) of SUDAF environment, the precomputed sharing relationships in $saggs_2(X)$ are exploited to reuse cached aggregation states to compute new ones, and if an aggregation state s of a UDAF in query sequences cannot be computed using cached ones, an aggregation state that can be reused for s will be computed and cached for later query executions. In SUDAF sharing environment, we prefetch a moment sketch (MS) [GDT+18, sta18] under one of the two selected query orders.

Our main findings are twofold. First, surprisingly, we observed that SUDAF without the sharing functionality 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 the sharing functionality 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 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 on PostgreSQL are only performed on the master node running PostgreSQL 11.4 (centralized environment).

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 TPCDS 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 [Ita15] and the third query model is evaluated on the TPC-DS [NP06] 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

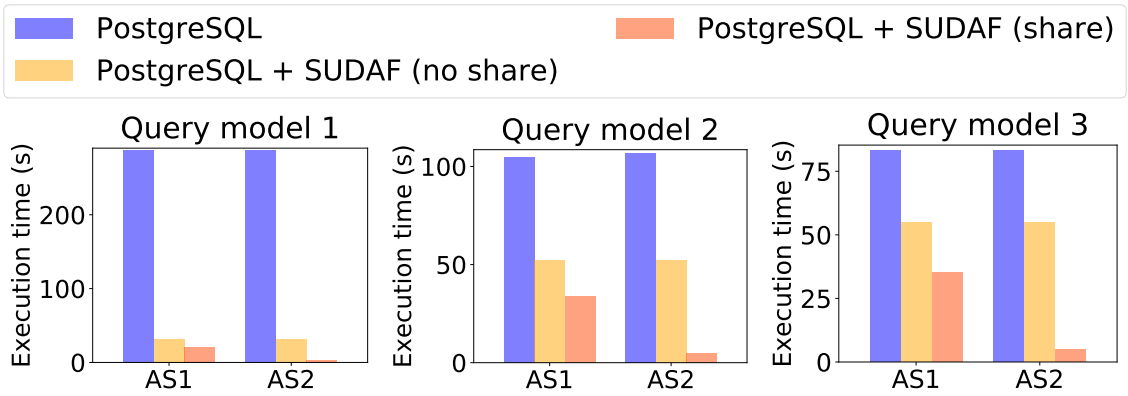

FIGURE 4.5: Total execution time of each query sequence in each query model (excluding queries with approximate median).

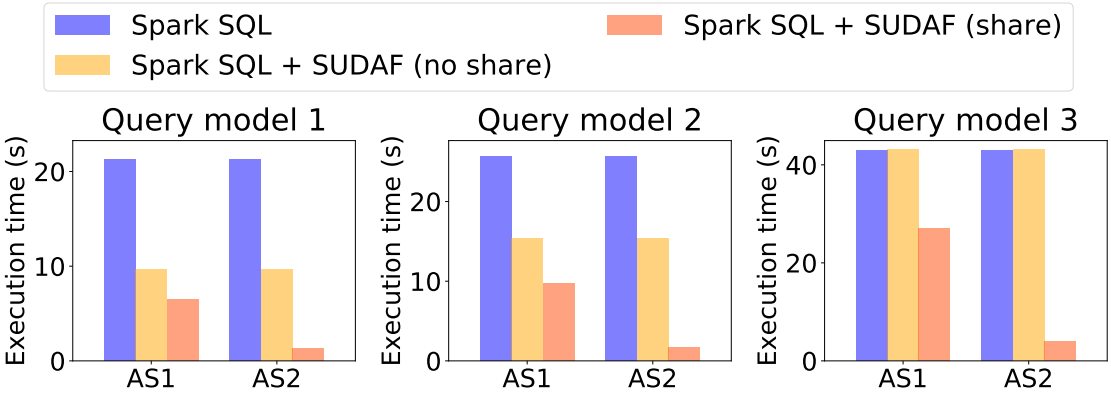

FIGURE 4.6: Total execution time of each query sequence in each query model (excluding queries with approximate median).

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 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) [GDT+18,sta18] 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 quantile, 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.

The total execution time of each query sequence in each query model is presented in Figure 4.5 for the case of PostgreSQL and in Figure 4.6 for the case of Spark SQL. Unsurprisingly, 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. 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 4.7 for the case of PostgreSQL and in Figure 4.8 for the case of Spark SQL.

SUDAF without sharing. For the case of PostgreSQL, compared to PostgreSQL UDAF queries, SUDAF speeds up UDAF queries up to 20X in query model 1, 4X in query model 2, and 2X in query model 3, which can be observed in Figure 4.7. For the case of Spark SQL, compared to Spark UDAF queries, SUDAF speeds up UDAF queries up to 3X in query model 1, 2X in query model 2, and have identical query time in query model 3, which is shown in Figure 4.8. The various performance improvements come from the size of inputs to be aggregated, i.e., query model 1 has the highest number of values to be aggregated, while query model 3 has the smallest number of values as aggregation input. 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.

SUDAF with sharing. SUDAF shares the computation results of partial aggregations in every query sequence. For the sequence AS1, we observe in Figure 4.7 (a), (c) and (e) and in Figure 4.8 (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 is able to reuse cached results from earlier aggregates in the sequence AS1. As it can be observed in Figure 4.7 (b), (d) and (f) and in Figure 4.8 (b), (d) and (f), the sequence AS2 is more advantageous for sharing due to the prefetched moment sketch. Indeed, the moment 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.

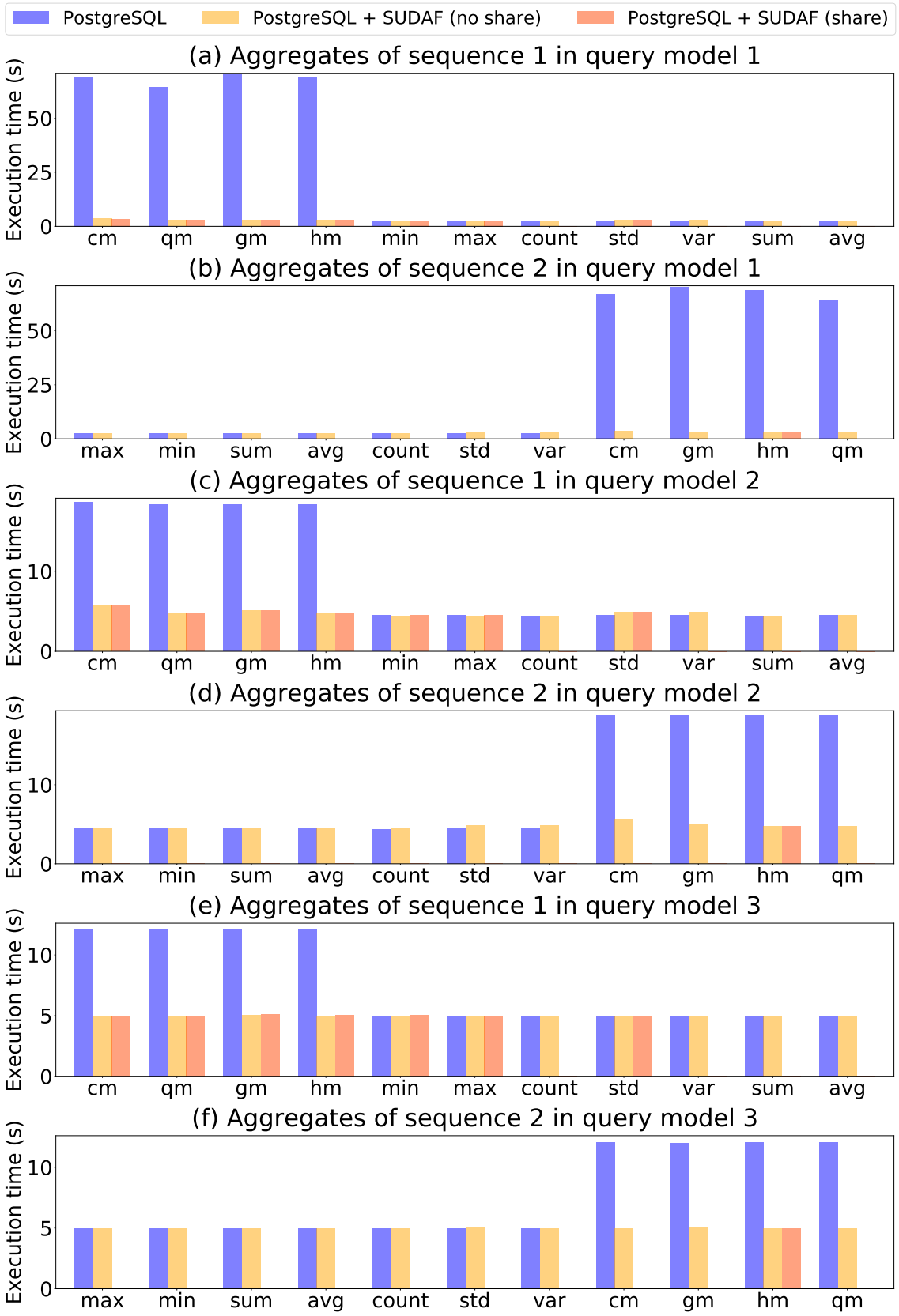

FIGURE 4.7: Execution time in PostgreSQL of each query in each query sequence.

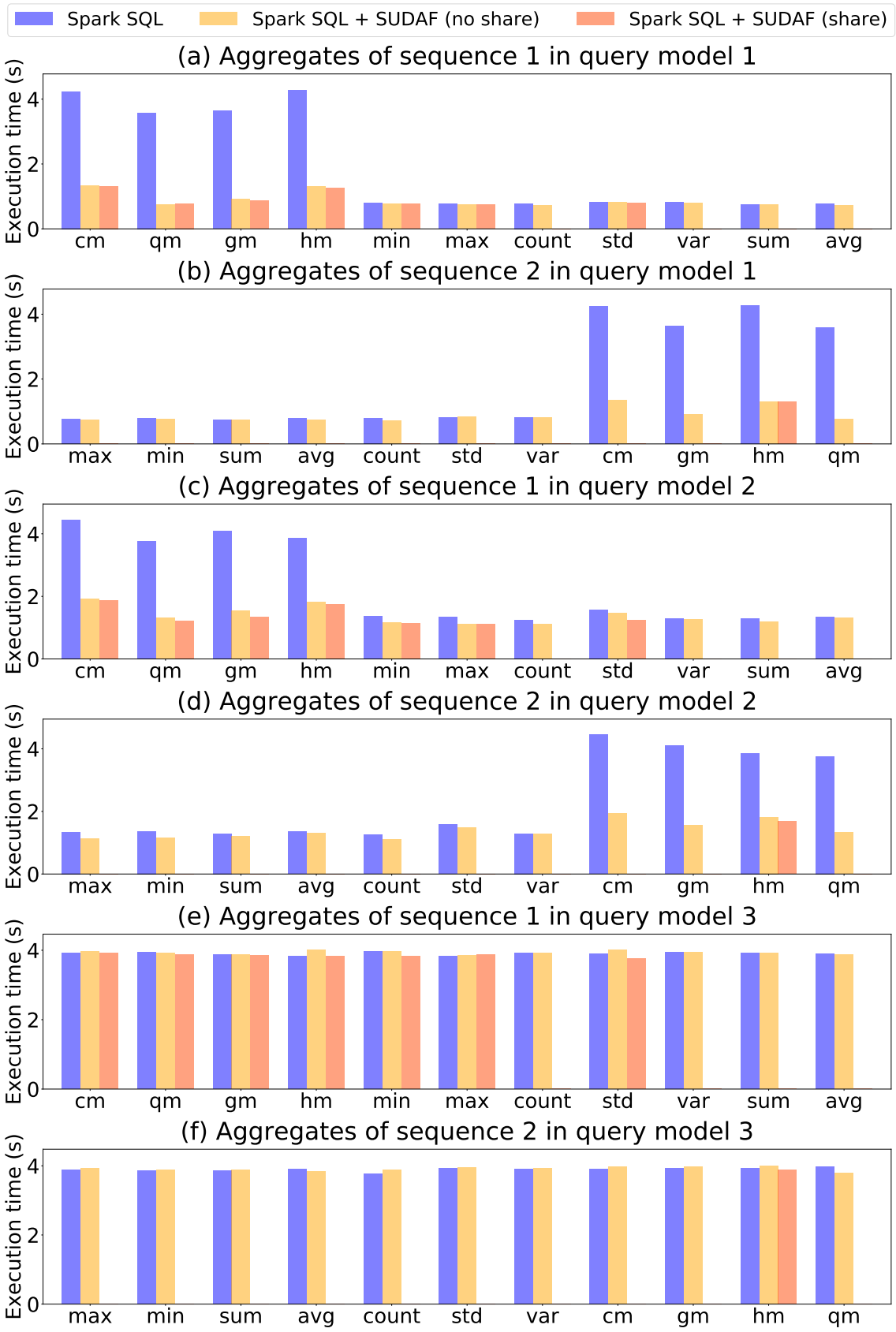

FIGURE 4.8: Execution time in Spark SQL of each query in each query sequence.

4.6 Summary

- Turning the sharing conditions into an algorithm could be cumbersome and also lead redundant computations. Thus, we propose to use symbolic representations of aggregation states to avoid the previous two issues.
- We propose the notion of symbolic aggregation state, which represents symbolic expressions of concrete aggregation states. Since an aggregation state can be an instance of several symbolic states, we propose the notion of natural instances, such that an aggregation state can be mapped to a natural instance of a symbolic state. Consequently, an aggregation state is uniquely mapped to a symbolic state.
- We define symbolic derivable sets of symbolic aggregation states to capture the sharing relationships of symbolic states. We found that, when an input is a multiset of positive values, symbolic derivable sets are equivalence relations in the symbolic space. Consequently, we only need to store natural instances of representatives, which is a unique symbolic state in an equivalent class.
- We discuss in detail the construction of sharing relationships of symbolic states. Specifically, we build a sharing digraph G for a symbolic space of symbolic states, where the length of the scalar functions of symbolic states is bounded by an integer l, which is a parameter controlled by users. We use the sharing conditions in Theorem 3 to identify sharing relationships and build G.
- We propose the 3D structure $(Rep, Chunks, NI)$ of SUDAF cache. Given an aggregation state s that is computed over one or several chunks of an input, we select for the computation of s in SUDAF cache, the representative of the symbolic state that s is a natural instance of in the dimension of Rep, the chunks that are covered by the computation of s, the natural instance of the corresponding representative.
- We propose a symbolic index to search a value in SUDAF caches for an aggregation state. Every entry of symbolic index is associated with a symbolic aggregation state in a symbolic space. We use a hash function to map an aggregation state to an entry in the symbolic index, and each entry is associative with its representative. We also identify and store a condition on the parameter sequence of symbolic states. Such a condition is used to select which natural instance of a representative can be used for computing an aggregation state.

Chapter 5

Prototype implementation

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. This prototype has been used in the experimental evaluation in Section 4.5. In this chapter, we present the implementation details of SUDAF prototype.

5.1 SUDAF architecture

Workflow overview. We present the general workflow of SUDAF in Figure 5.1. Generally, a user declares and registers a mathematical expression of a UDAF in SUDAF and then uses the UDAF in an SQL query. When SUDAF receives a query with UDAFs, it parses the query and mathematical expressions of UDAFs used in the current query. The expressions of UDAFs are parsed by a UDAF parser in SUDAF to construct aggregate expression trees (an aggregate expression tree simply represents the logical plan of computing an aggregation, details of which are present in Section 5.3). Then the UDAF optimizer applies rules to transform an aggregate expression tree and decomposes it to obtain partial aggregations. The caches are used by the UDAF optimizer to compute partial aggregations, and if some or all partial aggregations cannot be computed using caches, they are rewritten using system built-in functions and sent to an underlying system, e.g., Spark SQL or PostgreSQL.

Architecture overview. SUDAF prototype adopts a three-tier architecture presented in Figure 5.2 detailed as follows.

(1) The top layer is the UDAF interface. As depicted in Figure 5.2, end users can declare mathematical expression of aggregate functions at the UDAF interface and use them in SQL queries at the query interface.

(2) Advanced terminating function interface is put aside to create a T function that is not possible to be expressed using simple arithmetical operators supported in SUDAF, i.e., the moment solver used to estimate quantile [sta18, GDT+18]. Generally, one can define one or several aggregate functions and then write a program taking the defined aggregations as inputs. Such a 'plugged' program can be seen as a terminating function in the canonical form.

For example, one can define the following set of aggregation functions $SA(X) = \left(min, max, count(), \sum x_i, ..., \sum x_i^{k_1}, \sum ln(x_i), ... \sum ln^{k_2}(x_i)\right)$, then she can import the moment solver (MS) [GDT+18] to define an aggregation function $MS(SA(X), 0.5)$, which can be used to estimate median.

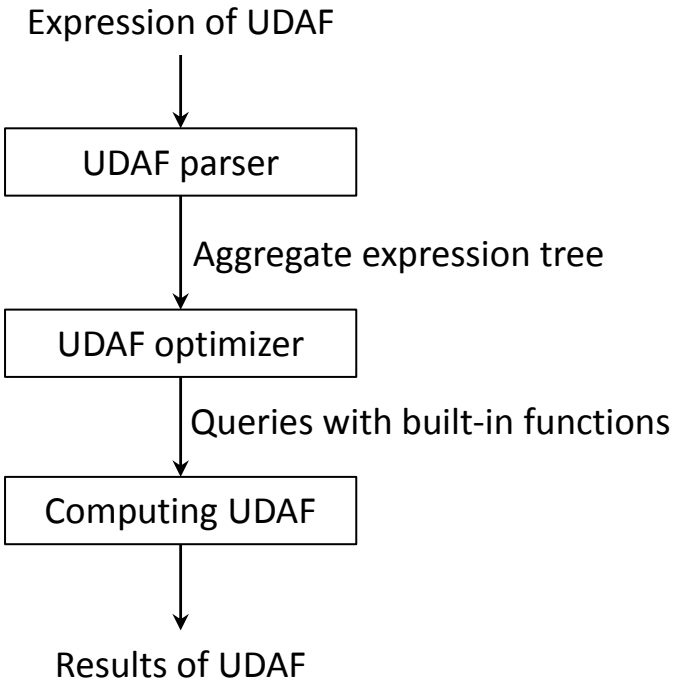

FIGURE 5.1: Workflow of processing UDAFs in SUDAF.

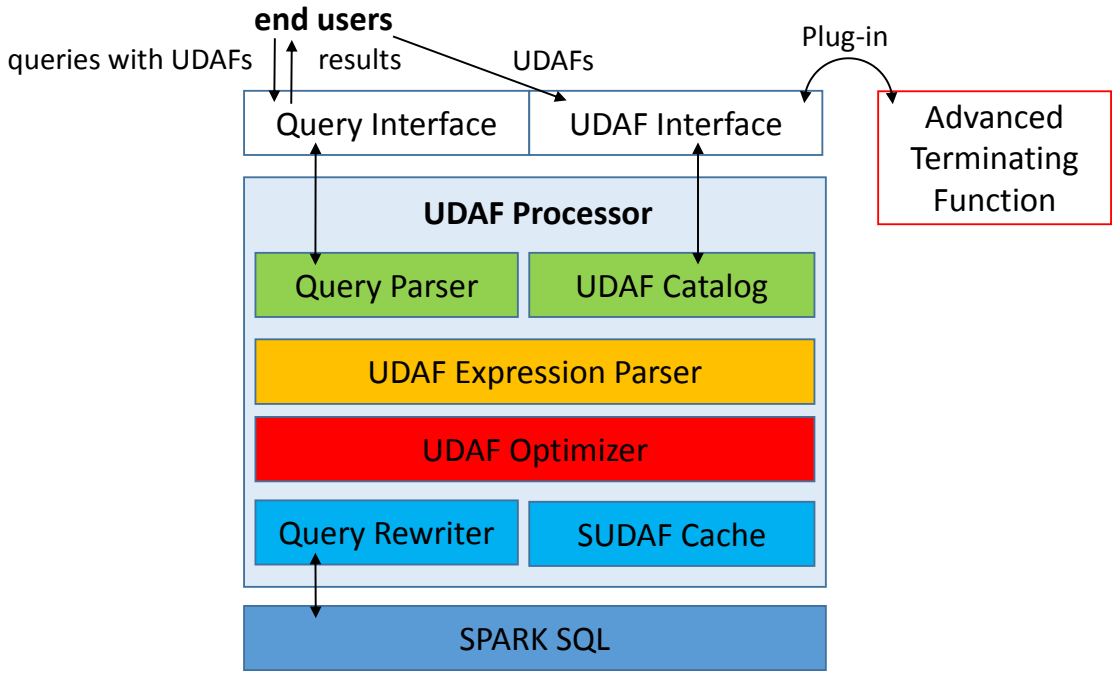

FIGURE 5.2: SUDAF prototype architecture.

(3) At the core of the system is a UDAF processor. The key components of the UDAF processor are (a) a UDAF catalog, (b) a query parser, (c) a UDAF expression parser, (d) an aggregation state optimizer, (e) a UDAF rewriter, and (f) SUDAF cache.

(4) SUDAF relies on an underlying system to compute queries, e.g., Spark SQL or PostgreSQL. Generally, SUDAF sends SQL queries with built-in functions to an underlying system.

UDAF processor. We present below the key components in the UDAF processor depicted in Figure 5.2.

- *UDAF Catalog*. At the time of defining a UDAF, SUDAF requires users to declare its mathematical expression and provide the name of the UDAF. UDAF catalog stores every declared UDAF as *'key-value'* pairs, where a *'key'* is a name of an aggregate function, and a *'value'* is the expression of the corresponding function.

- *Query Parser*. We rely on JSqlParser [onl] to parse a SQL query. Query Parser identifies the UDAFs and aggregate columns (columns that UDAFs are applied on) in a query.

- *UDAF Expression Parser*. When UDAF expression parser receives a mathematical expression of a UDAF α (a sequence of primitive functions), it constructs for α an AET (aggregate expression tree), which can be seen as a logical plan of computing α. The AET of α will be sent to the UDAF optimizer.

- *UDAF Optimizer*. UDAF Optimizer moves *a tuple-wise scalar computation* to *a final scalar computation* by applying transformation rules on aggregate expression trees, e.g., $\sum 4x_i \rightarrow 4(\sum x_i)$ (we present more details in Appendix A.1). Then, it decomposes the transformed AET to obtain the corresponding canonical form and aggregation states. At last, it reuses SUDAF cache to compute obtained aggregate states.

- *SUDAF Cache*. SUDAF Cache contains symbolic index and cached results. It has a 3D structure $(Rep, Chunks, NI)$. For an aggregation state s, SUDAF computes a hash code (Section 4.4) and extracts a constant sequence from s. The hash code is taken to get an entry point in a symbolic index, which can be seen as selecting a representative in the dimension of Rep of SUDAF 3D Cache. The constant sequence is used to get a cached aggregation state under a representative, which can be seen as selecting a natural instance in the dimension of NI of the 3D Cache. We rely on previous techniques, i.e., [DRSN98, WWDI17] to identify chunks covered by a current query. If all aggregation states in a current UDAF can reuse caches, then SUDAF computes the terminating function T of the current UDAF with these aggregation states as inputs and return the final result of the UDAF. If there exists at least one aggregation state that cannot be computed using caches, SUDAF evaluates the UDAF by scanning base data. In this case, we only send aggregation states, which cannot be computed using caches, to Query Rewriter, which will rewrite the current query to launch computation.

- *Query Rewriter*. Query Rewriter rewrites received queries with UDAFs to one with built-in functions which are aggregation states of UDAFs. A special case is when an aggregation state contains $\prod$ operator. In this case, we transform a product operator to a summation operator, e.g., if $s(X) = \prod f(x_i)$, then $s(X) = e^{\sum ln(|f(x_i)|)} \times (-1)^m$ where m is the number of negative $f(x_i)$.

5.2 SUDAF API

In this section, we present a scenario where an end-user creates a geometric mean in SUDAF and uses it in an SQL query. Knowing that the formula of geometric mean is $gm(X) = (\prod x_i)^{1/count}$. Then geometric mean can be defined by an end user in SUDAF using the following line of Scala code:

```
val gm = new UDAF.left.prod.right.^.left.cons(1.0)./.count.right
```

Using SUDAF UDAF interface to implement geometric mean is much compact compared to the way of using the counterpart one in Spark SQL (see below for a snippet of Scala code for implementing geometric mean in Spark SQL [dat19], which has the equivalent meaning as the previous implementation in SUDAF).

```
class GeometricMean extends UserDefinedAggregateFunction {
    override def inputSchema: org.apache.spark.sql.types.StructType =
       StructType(StructField("value", DoubleType) :: Nil)

    override def bufferSchema: StructType = StructType(
       StructField("count", LongType) ::
       StructField("product", DoubleType) :: Nil
    )

    override def dataType: DataType = DoubleType
    override def deterministic: Boolean = true

    override def initialize(buffer: MutableAggregationBuffer): Unit = {
       buffer(0) = 0L
       buffer(1) = 1.0
    }

    override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
       buffer(0) = buffer.getAs[Long](0) + 1
       buffer(1) = buffer.getAs[Double](1) * input.getAs[Double](0)
    }

    override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
       buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
       buffer1(1) = buffer1.getAs[Double](1) * buffer2.getAs[Double](1)
    }

    override def evaluate(buffer: Row): Any = {
       math.pow(buffer.getDouble(1), 1.toDouble / buffer.getLong(0))
    }
}
```

Secondly, a user can use the defined geometric mean in SQL queries. Assuming geometric mean is registered in UDAF catalog of SUDAF with the name 'myGM', and it is used in the following query:

```
SELECT myGM(A) FROM T;
```

When SUDAF receives this query, it finds the mathematical expression of geometric mean by its name myGM in the UDAF catalog. Then, the expression is parsed to generate an

aggregate expression tree, which is decomposed to generate canonical form and the aggregation states $\prod x_i$ and *count*.

If the caches cannot be used to compute the aggregation states, then the original query is rewritten as follows,

```
SELECT exp(sq1.agg1 / sq1.agg2) * pow(-1, MOD(sq2.m,2))) ^ (1/sq1.agg2)
FROM   (SELECT sum(ln(abs(A))) agg1, count(A) agg2 FROM T) sq1
       (SELECT count(*) m FROM T WHERE A < 0) sq2;
```

The rewritten query will be sent to Spark SQL to launch the computation, and its result is returned to a user.

If the two aggregation states, $\prod x_i$ and *count* can be computed using caches, then SUDAF simply takes their computation results as the input of T in the canonical form myGM to compute the geometric mean. Assuming that, only *count* can be computed using the cache, and $count(A) = n$ (at this step n can be seen as a constant value). Then, the original query will be rewritten as follow:

```
SELECT exp(sq1.agg1 / n) * pow(-1, MOD(sq2.m,2))) ^ (1/n)
FROM   (SELECT sum(ln(abs(A))) agg1 FROM T) sq1
       (SELECT count(*) m FROM T WHERE A < 0) sq2;
```

The above inner query only computes the aggregate function $\sum ln(|x_i|)$, which is the transformed shape of $\prod x_i$.

5.3 Generating canonical forms from mathematical expressions

In this section, we present how to obtain a canonical form for an aggregation. As explained previously, it is not realistic to expect users to define an aggregation function in the $(F, \oplus, T)$ framework, such that, we deal with the following problem in this section: *how to automatically generate a canonical form from a mathematical expression of an aggregation function*.

5.3.1 Expression model of aggregation functions

In order to solve the previous problem, we provide primitive functions and composition operators which can be used to construct mathematical expressions of aggregation functions.

We identify the following three general sets of primitive functions based on the number of their input arguments (it should be noteworthy that the three sets that we mentioned in Section 3.3 contains fixed classes of operators, but in this section we present the following sets which can contain arbitrary class of operators as long as an operator satisfy the property of a class).

- *Primitive scalar functions*: This class, denoted PS (primitive scalar), contains arbitrary scalar functions. *As long as a function is a scalar function, then it is an element in* PS, i.e., $f(x) = ln(x)$ can be an element in PS.
- *Primitive binary functions*: This class, denoted PB (primitive binary), contains arbitrary binary functions. *As long as a function is a binary function, then it is an element in* PB, i.e., arithmetical subtraction $-$ or arithmetical multiplication $\times$ can be an element in PB. Throughout this thesis, we use $\odot$ to denote a binary function in PB. We assign a binary function precedence, a positive integer, for every binary function in PB, where the minimum precedence of a binary function is 1, and we use

the function $pre(\odot)$ to get the precedence of $\odot$. For example, if PB only contains $-$ and $\times$, then $pre(-) = 1$ and $pre(\times) = 2$.

- *Primitive aggregation functions*: This class, denoted PA (primitive aggregation), contains arbitrary associative and commutative aggregation functions. *As long as a function is an associative and commutative aggregation function, then it is an element in* PS, i.e., $\sum x_i$ and $\prod x_i$ can be elements in PA. Throughout this thesis, we use $\sum_{\oplus} x_i$ to denote an associative and commutative aggregation function in this class. For example, $\sum x_i$ is the case of $\oplus = +$ and $\prod x_i$ is the case of $\oplus = \times$.

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

Complex scalar functions. We observe that a complex scalar function can be obtained in two ways, applying the composition operator or binary functions to compose or combine scalar functions, which is detailed as follows.

- *Using the composition operator*. As a natural way to compose functions in mathematics, the *composition operator*, denoted $\circ$, can be used to create complex scalar functions from the primitive ones. The class of such functions is denoted PS°. A function $g \in PS^{\circ}$ can be expressed as a composition of primitive scalar functions, and the length of $g(x)$, denoted $|g|$, gives the number of primitive functions used in the definition of $g(x)$, i.e., if $g(x) = f_l \circ ... \circ f_1(x)$, with $f_j \in PS$, then $|g| = l$. For example, assuming x^2 and $ln(x)$ are primitive functions in PS, then we can construct $g(x) = x^2 \circ ln(x)$, which is equivalent to $(ln(x))^2$, and $g(x)$ is indeed an element in PS°, and $|g| = 2$.

- *Using binary functions*. In addition, more complex scalar functions can be expressed using binary functions to combine scalar functions. The set of such functions, i.e., scalar functions that contain at least one binary operation, is denoted $PS^{\odot}$. Note that, we can use a binary function to combine two scalar functions, each of which can be from either PS, or PS°, or $PS^{\odot}$. For example, assuming x^2, x^3 and $ln(x)$ are primitive functions in PS and $+$ is a binary function in PB, then we can construct $g(x) = x^2 \circ ln(x) + x^3$, by using $x^2 \circ ln(x)$ from PS° and x^3 from PS, and $g(x)$ is indeed an element in $PS^{\odot}$.

Complex aggregation functions (UDAFs). Similar to the case of constructing complex scalar functions, we allow using the composition operator and binary functions between primitive aggregation functions from PA to create complex aggregation functions, which is detailed below.

- *Using the composition operator*. We allow to use the composition operator $\circ$ between a scalar function which can be from PS, or PS°, or $PS^{\odot}$, and an aggregation function to define a more complex UDAF. More precisely, in this context, the composition operators can be used in two ways: *(i)* to apply a scalar function on an output of a primitive aggregate function, i.e., $x^2 \circ \sum x_i$ which is equivalent to $(\sum x_i)^2$, or *(ii)* to apply a primitive aggregation on a set of data transformed using a scalar function, i.e., $\sum x_i \circ x_i^2$ which is equivalent to $\sum x_i^2$ (we can also omit the identity function of a primitive aggregation function, i.e., the example can be simply expressed $\sum \circ x_i^2$.). The class of such functions is denoted as PA°. For example, $(\sum x_i)^2$ and $\sum x_i^2$ can be elements in PA°.

- *Using binary functions*. More complex UDAFs can be expressed using primitive binary functions to combine several aggregation functions. The class of such functions is denoted $PA^{\odot}$. Note that, we can use a binary function to combine two aggregation functions, each of which can be from either PA, or PA°, or $PA^{\odot}$. For example, $\frac{\sum x_i}{n}$, $\frac{\sum x_i^2}{n}$ and $\frac{\sum x_i^3}{n} \times \sqrt{\sum x_i^2}$ can be elements in $PA^{\odot}$.

Scalar functions may simultaneously satisfy the property of PS, PS° and $PS^{\odot}$, e.g., $x^2 \circ ln(x)$ can be from PS or PS° since $x^2 \circ ln(x)$ is a scalar function. Similarly, an aggregation function can satisfy the property of PA, PA° and $PA^{\odot}$. For such a situation, we can make a decision based on whether a composition operator or a binary function is used in an expression of a scalar function or an aggregation function. For example, in the expression of $x^2 \circ ln(x)$, there is a composition operator, then we will consider both x^2 and $ln(x)$ are from PS and $x^2 \circ ln(x)$ is from PS°. For another example, in the expression of $\sqrt{x} \circ \sum \circ x_i^2$, which is equivalent to $\sqrt{\sum x_i^2}$ and is indeed an associative and commutative aggregation function, there are two composition operators, then we consider $\sum x_i$ is from PA and $\sqrt{x} \circ \sum \circ x_i^2$ is from PA°.

To be summarized, we model the construction of an expression of a UDAF as picking arbitrary elements from primitive scalar functions and primitive aggregation functions and combining them using the composition operator or primitive binary functions.

5.3.2 Mapping mathematical expressions of UDAFs into canonical forms

We explain below how to derive a canonical form from an expression of an aggregation function. W.l.o.g, we consider the most general expression of an aggregation function. Such functions are expressed using a terminating functions $T' \in PS^{\odot}$ applied on compositions, using binary functions in PB, of aggregate functions from PA° and have the following general form:

$$\alpha(X) = T'(\underbrace{(f_1' \circ \sum_{\oplus_1} \circ f_1(x_i)) \odot_1 ... \odot_{k-1} (f_k' \circ \sum_{\oplus_k} \circ f_k(x_i))}_{\text{an aggregation function in } PA^{\odot}}),$$

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

$$\begin{aligned} &F = (f_1, \ldots, f_k);\\ &\oplus = (\oplus_1, \ldots, \oplus_k);\\ &T = T'\big((f_1' \circ \sum_{\oplus_1} \circ f_1) \odot_1 \ldots \odot_{k-1} (f_k' \circ \sum_{\oplus_k} \circ f_k)\big). \end{aligned}$$

Example 12. *Geometric mean can be expressed as* $gm(X) = (\prod \circ x_i)^{\wedge}(1/count)$*, which corresponds to the following canonical form:*

$$F = (x_i, 1), \oplus = (\times, +), T = (\prod \circ x_i)^{\wedge}(1/count).$$

Parsing expressions of UDAFs. In the sequel, we discuss how to automatically generate a canonical form from a mathematical expression of a UDAF. Our approach can be summarized as the following two steps (see Figure 5.3):

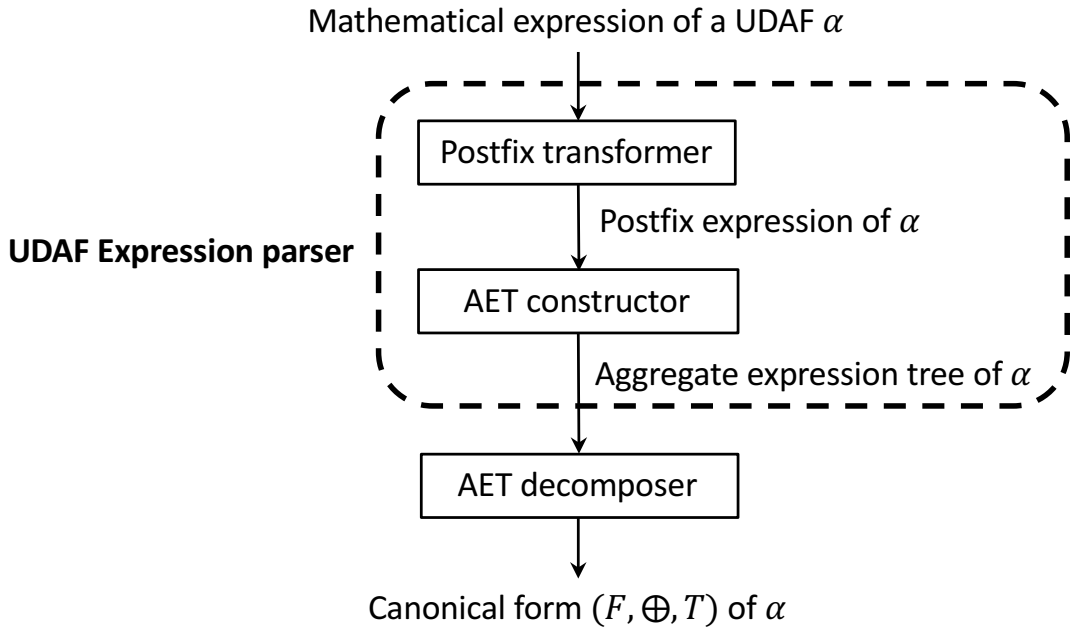

FIGURE 5.3: Parsing expressions of UDAFs to generate canonical forms of UDAFs.

Step 1: Every mathematical expression of a UDAF is parsed to construct an *aggregate expression tree* (AET), which can be seen as a logical plan of computing a UDAF.

Step 2: An AET is decomposed to obtain a canonical form.

We explain each of the two steps in detail in the following two sections.

5.3.3 Parsing mathematical expressions of UDAFs

In this section, we present the aggregate expression tree (AET) of UDAFs and explain how to parse a mathematical expression of a UDAF to construct an AET.

Aggregate expression tree. AETs are binary trees, and a node in an AET represents a primitive function from $PS \cup PB \cup PA$. We denote a node which represents a function from PS as a PS node, similar for functions from PB and PA. In an AET, PB nodes can have two children, and if a function is declared at the left side of a $\odot$ from PB, then the corresponding node is the left child of $\odot$ node, similar for a function declared at the right side of $\odot$. For instance, given an expression $x^2 + 3x$, then x^2 node and $3x$ node are respectively the left and right child of $+$ node. While, PS or PA nodes can only have one child, and if a function is composed with another function, the latter one is a child (left child as default) of the former one. For instance, given an expression of $3x \circ x^2$, then x^2 node is the left child of $3x$ node. Given another expression $\sum \circ x_i^2$, then x_i^2 node is the left child of $\sum$ node. Consequently, the evaluation order of primitive functions in a UDAF, i.e., which function should be first computed, is naturally represented by an AET in a bottom-up manner. We present the AET of geometric mean in Figure 5.4 (a), where the node labeled with 1 represents the constant function $f(x) = 1$.

Constructing an aggregate expression tree. We take a mathematical expression of a UDAF as an input to generate a corresponding AET. The given mathematical expression of a UDAF is a sequence of primitive functions, where each one is an element from one

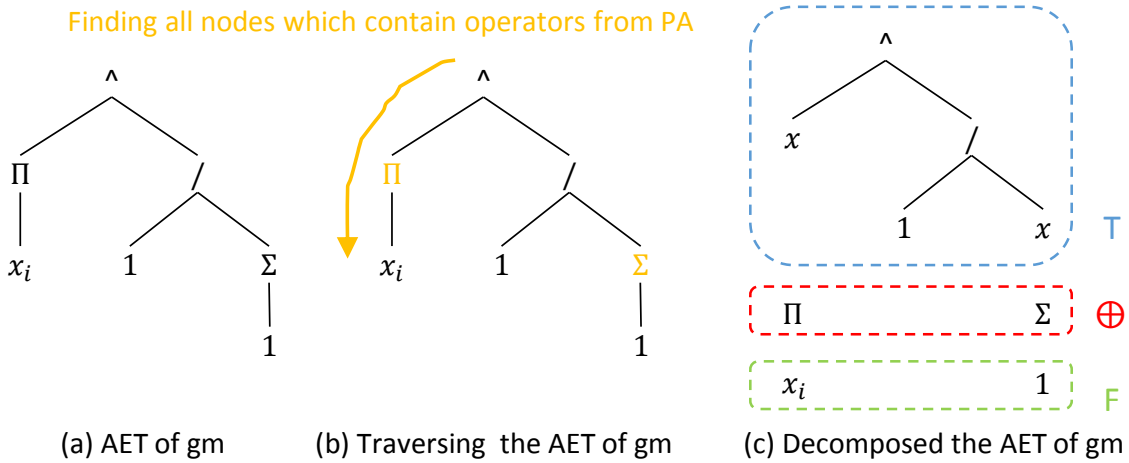

FIGURE 5.4: Aggregate expression tree (AET) of geometric mean and the corresponding decomposed AET.

of the following sets PS, PB, PA, and $\{\circ, LeftBracket, RightBracket\}$. For example, the expression of geometric mean, $(\prod \circ x_i)^{\wedge}(1/count)$, is a sequence of 13 primitive functions:

$$(LeftBracket, \prod, \circ, x, RightBracket, \wedge, LeftBracket, 1, /, \sum, \circ, 1, RightBracket), \quad (5.1)$$

where *count* can be expressed $\sum \circ f(x_i)$ with a scalar function $f(x) = 1$.

The input sequence of primitive functions present the *infix expression* of a UDAF, which is a natural way of writing a mathematical expression. In order to generate an AET, an infix expression is transformed into a corresponding *post-fix expression*, where a binary function, or a composition operator, is behind its two operands (in our context, an operand of a binary function can be a scalar function or an aggregation function), i.e., the post-fix expression of $x^2 + ln(x)$ is $x^2\ ln(x)\ +$, and the post-fix expression of $\sum \circ x^2$ is $\sum\ x^2\ \circ$. We continue the previous example of geometric mean, the sequence of primitive operators in equation (5.1) represents the infix expression of geometric mean, and it can be transformed into the following sequence of primitive operators, which represents the post-fix expression of geometric mean:

$$(\prod, x, \circ, 1, \sum, 1, \circ, /, \wedge). \quad (5.2)$$

Figure 5.4 (a) depicts the AET constructed from the post-fix expression (equation (5.2)) of geometric mean.

The details of transforming an infix expression to a postfix expression are shown in algorithm 1, which follows the general structure of the shunting-yard algorithm [Dij], and algorithm 2 shows how to construct an AET from a post-fix expression of a UDAF.

5.3.4 Decomposition of aggregate expression tree

In this section, we present algorithm 3 to decompose an AET. We explain the details as follows: we execute a preorder traversal on an AET to find all nodes which contain primitive functions from PA, where we decompose an AET into three parts corresponding to the three elements in a canonical form $(F, \oplus, T)$. More precisely, if a node in an AET contains a function $\sum_{\oplus_i}$ that is from PA, denoted as $\sum_{\oplus_i}$ node, then the function $\oplus_i$ will be added in the $\oplus$ function of a corresponding canonical form. The child of the $\sum_{\oplus_i}$ node is the root of an expression tree of a scalar function $f_i(x)$, and it will be added into the F function of the canonical form. We also change the $\sum_{\oplus_i}$ node to be a node containing

the identity function and remove its child in an AET. When all $\sum_{\oplus_i}$ nodes have been processed according to the above procedure, we obtain the F function and the $\oplus$ function in a canonical form. Consequently, the left AET, which has been modified, represents the T function in a canonical form. For example, Figure 5.4 (b) shows the preorder traversal on the AET of geometric mean, and Figure 5.4 (c) presents the decomposed AET of geometric mean and the obtained canonical form.

Algorithm 1: Transforming an infix expression of a UDAF to a corresponding postfix expression

```
Input: infix, an array of operators representing an infix expression of a UDAF
Output: postfix, an array of operators representing a postfix expression of a UDAF
Algorithm Infix2Postfix(infix)
    var OperatorStack os ;
    var Operator tp ;
    var OperatorArray post-fix ;
    for i := 0 to infix.length do
        tp ← infix[i];
        if tp.type == PS or tp.type == PA then
            postfix.add(tp);
        else if tp.type == PB or tp.type == ∘ then
            while os.empty == false and predence(tp) ⩽ predence(os.peek) do
                postfix.add(os.pop);
            end
            os.push(tp);
        else if tp.type == LeftBracket then
            os.push(tp);
        else                                              ▷ tp is RightBracket
            while os.empty == false and os.peek.type ≠ LeftBracket do
                postfix.add(os.pop)
            end
            os.pop;
    end
    while os.empty == false do
        postfix.add(os.pop);
    end
    return postfix;
    end;
proc predence(op)
    if op.type == PB then
        return pre(op); ▷ pre(op) is a function to get a precedence of a function in PB
    end
    if op.type == ∘ then
        return PB.MP + 1;  ▷ PB.MP is the maximum precedence of functions in PB
    end
    return -1;
    end;
```

Algorithm 2: Constructing an aggregate expression tree from a post-fix expression of a UDAF

Input: postfix, an array of operators representing a postfix expression of a UDAF
Output: root, the root node of an aggregate expression tree
Algorithm `ConstrctAET(`*post-fix*`)`

```
var NodeStack ns;
var Node root;
var Operator tp;
var Node node1;
var Node node2;
for i := 0 to postfix.length do
    tp ← postfix[i];
    if tp.type == ∘ then
        node1 ← nodeStack.pop;
        node ← nodeStack.pop;
        node2 ← node;
        while node2.leftChild ≠ null do
            node2 ← node2.leftChild;
        end
        node2.leftChild ← node1;
        nodeStack.push(node);
    else
        var Node node;
        if tp.type == PS  or tp.type == PA then
            node.operator ← tp;
            ns.push(node);
        else                                                    ▷ tp ∈ PB
            node.operator ← tp;
            node.rightChild ← ns.pop;
            node.leftChild ← ns.pop;
            ns.push(node);
    end
end
root ← ns.pop;
return root;
end;
```

Algorithm 3: Decomposition of aggregate expression tree

Input: AET, the root node of an aggregate expression
Output: WFA, $(F, \oplus, T)$ in well-formed aggregation
Algorithm `DecomposeAET(`AET`)`

```
var NodeStack ns1;
var WellFormedAggregation WFA;
var NodeList F;
var OperatorList ⊕;
var Node node1;
ns1.push(AET);
while ns1.empty == false do                    ▷ Pre-order traversal on AET
    node1 ← ns1.pop;
    if node1.operator.type == PA then          ▷ Identifying a PA node
        var Node node2 ← node1.leftChild;
        F.add(node2);
        ⊕.add(node1.operator);
        node1.operator ← identityFunction;
        node1.leftChild ← null;
        continue;
    end
    if node1.rightChild ≠ null then
        ns1.push(node1.rightChild);
    end
    if node1.leftChild ≠ null then
        ns1.push(node1.leftChild);
    end
end
WFA ← (F, ⊕, AET);
return WFA;
end;
```

Chapter 6

Related works

There is a wealth of researches on optimizing queries with aggregate functions, aggregate query rewriting using views, sharing computations for aggregate queries, caching results of aggregate queries, distributed computations of aggregate functions. Earlier works focus on standard or predefined aggregate functions (e.g., [YbL95, CD97, GCoS+97, CNS99, CN05, HGHS07]) and then extended to UDAFs (e.g., [Coh06, HPK+13, CTK+16, KBY17]). We categorize the opportunities for optimizing queries with aggregations in the following two scenarios: processing a single query and a workload of several queries.

- In the scenario of processing a single query, partial aggregation appeared as an important technique used to improve the performance of aggregate functions: instead of computing aggregation on a complete multiset, applying aggregation on subsets of the multiset and merging intermediate results is an efficient solution in various situations. We study related works in this category in Section 6.1.
- Using materialized views or caches to accelerate queries with aggregation is also well studied in the scenario of processing a workload of several aggregate queries or a case of repetitive data analytic. Most of these works focus on identifying the reusing opportunities for aggregate queries over various data granularity. Several works also study the opportunities in the computation dimension, i.e., computing an aggregate function over another aggregate function. The related works in this category are studied in Section 6.2.

6.1 Partial aggregation

6.1.1 Optimizing queries with aggregate functions

Aggregate functions are applied over a multiset of values, which is generally computed after a group-by operator in a query execution plan. Because a group-by operator has the ability of duplicate elimination, such that applying group-by operator before join can help reduce the input size of join, which is usually referred to as aggregation push down. In order to ensure the correctness of query results, group-by with partial aggregations are pushed before join instead of a final aggregation function.

An early overview pertaining to optimizing queries with standard aggregate functions can be found in [SF95]. We briefly discuss results related to applying partial aggregation. This line of research is started independently in [YL94] and [CS94]. In [YL94], they proposed the necessary and sufficient condition that an expression of relational algebra need to satisfy for performing a group-by operator before join in the case of queries with standard aggregate functions (MIN, MAX, SUM, COUNT, AVG), and they extended the condition to more general functions with an algebraic property (decomposable aggregation, which is similar to partial aggregation). Their later work [YbL95] also studied the possibilities of computing a group-by after join, since this may help reduce the number of

rows in a group. The main transformations proposed in these early two works are eager aggregation (performing group-by operator before join) and lazy aggregation (performing group-by operator after join), which are two inverse orders of computing group-by operator and join. The class of eager aggregation also contains three sub-classes: eager group-by, eager count, and double eager, where the last one is the combination of applying the first two together. Aggregation used in a query must satisfy the decomposable property to obtain an equivalent expression of the query using eager group-by transformation, because after join partial aggregate values in groups with smaller granularity have to be merged into a group with a bigger granularity. The eager count transformation is simply used to compress duplicate values in a group. When applying these two transformations together, an expanding function is required to compute the final aggregate values using duplicate partial aggregate values. Independently, the opportunity of including a group-by operator in query optimization was also studied in [CS94], and they proposed three kinds of group-by transformations: (1) invariant grouping, (2) simple coalescing grouping and (3) generalized coalescing grouping. They also discuss a cost-based solution to apply the transformations. The invariant grouping transformation does not require any specific properties of aggregate functions, but it can be only applied when relations are joined on foreign keys, which also need to be grouping attributes in a query since no duplicate values need to merge after join. While aggregate functions require satisfying the associativity and commutativity to apply simple coalescing grouping. More transformation rules for aggregate queries were studied in [GHQ95], where coalescing groups (which they call generalized projection) also require partial aggregation.

The optimizing technique for correlated subqueries with aggregations are unified with the reordering of join and group-by aggregation in [GLJ01]. The execution plan of an original correlated subquery with aggregation is to take every row of an outer query and compute an aggregate value in a subquery, which is a row-oriented strategy. The correlated subquery can be removed by rewriting into another formulation applying an outer join and aggregation [Day87], then reordering of group-by aggregation and an outer join can be evaluated by a cost-based solution to generate an efficient execution plan [GLJ01], which also requires partial aggregation.

A later work [Coh06] extended eager group, eager count and double eager transformations, which are proposed in [YbL95], to queries with user-defined aggregate functions. A canonical form of user-defined aggregation is developed to systematically obtain partial aggregation in order to leverage previous aggregation push-down (computing aggregation before join) techniques. Unlike previous works where algebraic properties are restricted over aggregation to have partial aggregation, the canonical form of user-defined aggregations captures the construction of a user-defined aggregation, which is a more general solution to obtain partial aggregation.

Compared to these works, we do not deal with the reordering of group-by aggregation and join, or propose a cost-based solution to identify when to apply the reordering. While we concentrate on how to systematically and automatically generate partial aggregation from a mathematical expression of aggregations [ZTG17b, ZTG17a, ZTG18, ZT19]. Since partial aggregation is the essential technique required for the reordering of group-by and join, we argue that our approach can be integrated with existing solutions to optimize queries with UDAFs automatically.

6.1.2 User-defined aggregate function API

Most modern data management and analytical systems support UDAFs (e.g., [apaa,apad, apab,RDBc,RDBa,RDBb]). In original MapReduce (*MR*) framework [DG04,apac], 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 [HPK+13]). However, in most of the recent systems, users define UDAFs using an *IAME* pattern [Coh06] (Initial values, Accumulating functions, Merging functions and Evaluating functions). Although such an approach enables one to exploit the properties of the merging functions to allow optimization based on partial aggregation, e.g., parallel computation of the merging functions, part of the UDAF semantics is still hidden in the implementation which hampers some optimization opportunities such as aggregate sharing. In addition, implementing UDAF in existing frameworks may be a tedious task since it is up to the user to map its UDAF to the implementation paradigm (*MR* or *IAME*). In [Coh06], a generic UDAF framework is proposed where partial aggregation can be systematically derived from a well-formed expression of UDAFs. We build on these later works 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 $IAME$ pattern. Consequently, with SUDAF, a user does not need to handle the problem of how to obtain partial aggregation from UDAFs. Moreover, SUDAF knows the semantics of partial aggregation (primitive operators used in partial aggregation) which extends the optimization opportunities [ZT19].

6.2 Caching and materializing queries with aggregation

6.2.1 Rewriting aggregate queries using aggregate views

Evaluating an aggregate query using a materialized aggregate view other than base relation can be significantly efficient because the number of tuples in an aggregate view is usually several orders of magnitude less than the number of tuples in base relations. In order to obtain this efficient strategy, a query processor transforms an aggregate query and replaces a subquery with aggregations by a view with aggregations. Generally, a query processor needs to verify two problems.

- The first one is on whether the view is equivalent to or contained in the query, which is undecidable for arbitrary queries [AHV95]. Completely determining whether a conjunctive query can be rewritten using views is an NP-complete problem [LMS95]. Practical solutions apply syntactic comparisons between queries and views. A practical algorithm for rewriting queries with sum and count can be found in [CNS00], and a survey can be found in [Coh05].

- The second problem appears when the aggregation in the query is different from those in the view. In such a case, a query processor should know how aggregate values in the view can be combined to compute the aggregation in the query. Users can explicitly program such a sharing relationship of aggregations as a computation rule [Coh06] to deal with built-in aggregate functions. However, for general aggregation functions, especially UDAFs, previous solutions for rewriting aggregate queries using views in [GHQ95, CS96, SDJL96, CNS99, GL01, GT00, CNS06] do not consider the problem of how to identify such a relationship of aggregations. Consequently, given a query and a set of views, where the aggregation in the query is different from those in the views, previous solutions may not be complete. We deal with the problem of how to compute an aggregation (UDAF) from the others [ZT19], which can be merged with previous solutions to have more rewriting candidates in this line of research.

6.2.2 Caching aggregate queries

Different facets of caching aggregate queries have been studied e.g., reusing caches to accelerate multi-dimensional queries [CD97, DRSN98], or identifying overlapping parts for multiple aggregate queries with various selection predicates [HGHS07], group-by attributes [CN05] and sliding-windows [AW04, KWF06]. Most of these approaches focus on the data granularity dimension, i.e., they consider the problem of sharing the same computations across different ranges or granularity of data. Our work [ZT19] does not consider the data granularity dimension where existing techniques, e.g., [DRSN98, WWDI17], can be used to extend SUDAF in this direction. [CNS99, CNS06] enables sharing between different aggregations using predefined equivalence rules, while DataCanopy [WWDI17] caches results of basic aggregates (e.g., $\sum x, \sum x^2, \ldots$) and uses predefined decomposition rules to compute statistical measures from basic aggregates. SUDAF lies in this research direction, extending existing approaches with the ability to identify more sharing opportunities among different UDAFs dynamically.

Until the time of writing this thesis, based on our observation, the most related work is DataCanopy [WWDI17]. Therefore, we present the specific differences between DataCanopy and SUDAF [ZT19] as follows. Our approach is complementary to DataCanopy in the sense that DataCanopy deals with "sharing w.r.t. the data dimension" while SUDAF deals with "sharing w.r.t. the computation dimension". DataCanopy "caches the basic aggregates of statistical measures" and then is able to reuse them for future queries. Basic aggregates are maintained at a granularity of a chunk (smallest portion of data). DataCanopy allows sharing between queries over overlapping or partially overlapping chunks. However, in DataCanopy basic aggregates are fixed in advance ($\sum x_i$, $\sum x_i^2$ and $\sum x_i \times y_i$) and the decomposition of a given aggregate into basic ones is predefined (see Table 1 of [WWDI17]). SUDAF allows to automate the sharing possibilities with respect to the computation dimension, i.e., it allows sharing between different aggregates without using predefined decomposition rules. In addition, SUDAF does not rely on fixed basic aggregates but uses instead partial aggregates derived from executed queries.

Chapter 7

Conclusions

UDAFs (user-defined aggregation functions) are becoming a type of fundamental operations in advanced data analytic. UDAFs are generally defined by programming each function in the IAME framework, i.e., specifying initial functions, accumulating functions, merging functions and evaluating functions. Such a mechanism requires users to identify these functions from their UDAFs and to ensure that the identified merging function is associative and commutative. Modern data management systems require a declarative approach for defining UDAFs and the ability to synthesize partial aggregations automatically, which can relieve users' mental overhead. The current UDAF mechanism also has two severe drawbacks, which leads to the loss of opportunities to accelerate queries with UDAFs. Firstly, computing a UDAF using the current mechanism is much slower compared to system built-in functions. Moreover, a UDAF defined by the current mechanism is a black box to a query processor.

In order to overcome these issues, we present SUDAF in this thesis. We target on designing a declarative approach to define UDAFs, where we automatically generate efficient partial aggregations, rewrite partial aggregations using built-in functions, cache and reuse partial aggregations across various UDAFs. In this thesis, we first studied how to map aggregation functions, in a systematic way, into generic *MRC* algorithms, and we identified when aggregations can be efficiently executed on MapReduce-style platforms. We also discuss how to generate partial aggregations from mathematical expressions of UDAFs. Then we concentrated on introducing the design principles underlying SUDAF, a system that provides a set of predefined functions together with a composition operator to enable users to write their UDAFs by declaring mathematical expressions. SUDAF comes equipped with the ability to automatically generate parallel implementation from a mathematical expression of a UDAF and supports efficient dynamic caching and sharing of partial aggregations of UDAFs. We showed experimentally the benefit of sharing aggregates to improve the performances of queries with UDAFs.

We observe the following research perspectives that can be taken as future works of SUDAF.

- Our immediate future work is merging the computation dimension of SUDAF with the data dimension proposed in existing works [DRSN98, WWDI17]. Therefore, queries with various data, i.e., different range predicates, and various UDAFs can be computed over caches.

- One of the traditional techniques on optimizing join and group-by queries is aggregation push down. In such a scenario, a query processor needs to know for a UDAF partial aggregations, merging functions, and expanding functions (recovering the duplicate values caused by pushing down count). As explained in this thesis, SUDAF has the ability to automatically generate these functions (expanding

functions can be automatically synthesized by inferring $\oplus$ functions in the canonical form of SUDAF). Therefore, one of the future works is to investigate how SUDAF can help query processors to obtain query plans without users' explicit interference.

- As a future research direction, we envision to exploit the fact that the semantics of UDAFs is known by SUDAF to investigate query rewriting problems for join and group-by queries with UDAFs. Knowing that traditional approaches of query optimizations, rewriting queries using materialized views and caching query results mainly focus on built-in or predefined aggregate functions. We believe that, if a query processor can capture the semantics of UDAFs, it could identify how to compute a UDAF from others required in answering queries using materialized views or caches.

- In the problem of aggregate view selection, one usually wants to find some suitable views to materialize, such that they can be reused to answer more queries in a workload. In such a scenario, SUDAF can help identify the overlapping computations of aggregations in views, i.e., whether an aggregation in a view can be computed from another view.

- OLAP queries are computed over a data cube instead of base data. Specifically, in every cell of a data cube, an aggregate summary for the smallest data granularity is computed and stored. Traditionally, these aggregate summaries are only reused to compute queries with built-in or predefined functions. Since SUDAF can capture the relationship of how to compute a UDAF from another one, we envision to investigate aggregate summaries in data cube to compute queries with UDAFs.

Appendix A

A.1 Moving a tuple-wise scalar computation to a final scalar computation

In this section, we explain the UDAF optimization principle in SUDAF, *moving a tuple-wise scalar computation to a final scalar computation*. W.l.o.g, assuming a UDAF α has the following shape $\alpha(X) = T(s(X))$ with an aggregation state $s(X) = \sum_{\oplus} f(x_i)$, and the input X has n tuples, i.e. $count(X) = n$. During the computation of α, we compute f for a value x_i in every tuple of X, which can be seen as a *tuple-wise scalar computation*. Consequently, the scalar function f is computed n times. If the UDAF α can be transformed into the following shape $\alpha(X) = T(f(\sum_{\oplus} x_i))$, where the scalar function f is moved outside the aggregation operator $\sum_{\oplus}$. Consequently, the scalar function f is computed only once, which can be seen as a *final scalar computation*. In a word, if we move a tuple-wise scalar computation to a final scalar computation for the computation of a UDAF α, we can avoid $n-1$ times of computing a scalar function.

Scalar pull-up transformations. To capture such an optimization opportunity, we propose the following 3 *scalar pull-up* (SPU) rules [1],

- SPU 1: $\sum f_2 \circ f_1(x_i) \rightarrow f_2(x) \circ \sum f_1(x_i)$, where $f_2(x) = ax$ with a constant a;
- SPU 2: $\prod f_2 \circ f_1(x_i) \rightarrow f_2(x) \circ \prod f_1(x_i)$, where $f_2(x) = x^a$ with a constant a;
- SPU 3: $\prod f_2 \circ f_1(x_i) \rightarrow f_2(x) \circ \sum f_1(x_i)$, where $f_2(x) = a^x$ with a constant a.

For example, using SPU 1, $\sum 3x_i^2$ can be transformed into $3(\sum x_i^2)$, which can avoid $n-1$ times of arithmetic multiplication since computing $\sum 3x_i^2$ requires computing $3x_i^2$ for every value in X, while $3 \times \sum x_i^2$ only needs to compute one times of arithmetic multiplication.

Splitting transformations. In Section 3.4.3, we propose two splitting rules to transform one aggregation state into two aggregation states. This kind of transformation also has the property of moving a tuple-wise scalar computation to a final scalar computation. We recall the splitting rules as follow:

- Split 1: $\sum(f_1(x_i) \odot f_2(x_i)) = \sum(f_1(x_i)) \odot \sum(f_2(x_i))$ with $\odot \in \{+, -\}$;
- Split 2: $\prod(f_1(x_i) \odot f_2(x_i)) = \prod(f_1(x_i)) \odot \prod(f_2(x_i))$ with $\odot \in \{\times, /\}$.

Using the above two rules, the $\odot$ operator only needs to be computed once instead of computing it for every tuple. For example, using the rule Split 1, $\sum 3x_i + x_i^2$ can be transformed into $\sum 3x_i + \sum x_i^2$, which can avoid $n-1$ times of arithmetic addition.

[1] We call these rules as scalar pull-up rules because a scalar function can be pulled up a primitive aggregation operator in an aggregate expression tree. The term 'scalar' is used to distinguish from the traditional pull-up transformations in aggregate query optimization.

Expression	Number of arithmetic addition	Number of arithmetic multiplication
$s(X) = \sum_{i=1}^{n}(x_i + a)^b$	$2n$	$n(b-1)$
$s(X) = \sum_{k=0}^{b}\binom{b}{k}a^{b-k}(\sum_{i=1}^{n}(x_i)^k)$	$nb + b$	$n(b-1) + (\frac{(b+2)(b-1)}{2} + b)$
$PowerSeries(X, b) = (\sum x_i, ..., \sum x_i^b)$	nb	$n(b-1)$

TABLE A.1: Cost of computing $\sum_{i=1}^{n}(x_i + a)^b$ and its binomial expression.

Binomial transformations. We can combine *scalar pull-up rules* and *splitting rules* to have *binomial transformations* of aggregation states. We explain this as follows. An aggregation state $s(X) = \sum(x_i + a)^2$, which is denoted as an original expression with a constant a and an exponent 2, can be trivially transformed to $s(X) = \sum(x_i^2 + 2ax_i + a^2)$. Then, we have the following shape of s by applying scalar pull-up rules and splitting rules: $s(X) = \sum x_i^2 + 2 \times a \times \sum x_i + count(X) \times a \times a$, which is denoted as the *binomial expression*. We also denote such a transformation as the *binomial transformation* over the original expression of s.

In the sequel, we focus our discussion on binomial transformations of aggregation states. As explained later, the major benefit of binomial transformation is that caches in SUDAF can be fully exploited to share computations, i.e., reusing caches to compute original expressions with various constants a and various exponents that are bigger than 2. However, a binomial expression is more expensive to compute than the corresponding original expression since a binomial expression contains more aggregation states that need to compute. Note that a binomial expression only needs to compute arithmetic addition and multiplication. We observe that every aggregation state in a binomial expression inevitably needs a summation operator. Consequently, the overhead of applying more arithmetic additions cannot be avoided. While for the computation of arithmetic multiplication, we show below an algorithm to compute binomial expression which requires an approximately same cost as computing the original expression.

In the following, we first compare the cost of computing an original expression and the corresponding binomial expression using the proposed algorithm. Then, we show binomial transformations bring more possibilities of sharing computations.

- *Cost analysis*. Our cost model is based on the number of (arithmetic) addition and (arithmetic) multiplication that is required in the computation of original expression and a binomial expression since these two basic operations are enough to execute their computations. We consider $count(X)$ is cached in system since $count(X)$ is a widely used in query optimization, and we let $count(X) = n$. We begin our analysis with the initial case, $s(X) = \sum(x_i + a)^2$, and then we present the results for a general case.

  Computing the original expression, $s(X) = \sum(x_i + a)^2$, needs $2n$ additions, i.e., computing $x_i + a$ for every x_i and a summation, and n multiplications, i.e., computing $(x_i + a)^2$ for every $(x_i + a)$. While, the transformed expression, $s(X) = \sum x_i^2 + 2 \times a \times \sum x_i + count(X) \times a \times a$, requires $2n + 2$ additions, i.e., computing two summations and 2 final additions, and $n + 4$ multiplications, i.e., computing x_i^2 for every x_i and 4 final multiplications. Such that binomial expression requires computing 2 more additions and 4 more multiplications than the original expression, which are approximately identical since n is quite large in general.

  We analyze a general case of binomial transformation in the following, where an original expression of an aggregation state is $s(X) = \sum(x_i + a)^b$ with a constant a and a positive integer b. We summarize the cost of computing the original expression and its binomial transformation in Table A.1, where the computation cost of

binomial coefficients are not taken into account, i.e., we consider every $\binom{b}{k}$ is a constant value. We explain their costs as follows. In the computation of the original expression, $s(X) = \sum(x_i + a)^b$, we need $2n$ additions, i.e., a first n for computing $x_i + a$ and a second n for computing a summation, and $n(b-1)$ multiplications, i.e., computing $(x_i + a)^b$ for every $(x_i + a)$. The original expression can be transformed into the following binomial expression, $s(X) = \sum_{k=0}^{b} \binom{b}{k} a^{b-k} (\sum_{i=1}^{n} (x_i)^k)$, which we use the algorithm 4 to compute. We analyze the cost of the proposed algorithm below. In algorithm 4, we need to compute a sequence of aggregation states, which is $(\sum x_i, \sum x_i^2, ..., \sum x_i^b)$ and is denoted as *PowerSeries(X,b)*. The subroutine to compute $PowerSeries(X, b)$ stores for every input value x_i the result of x_i^l and reuse it to compute x_i^{l+1}. Thus, it takes $n(b-1)$ multiplications in total (see Table A.1). While it still requires nb additions. Finally, computing the binomial expression takes $nb + b$ additions and $n(b-1) + (\frac{(b+2)(b-1)}{2} + b)$ multiplications. Since n is generally much larger than b, the costs of computing multiplication for original and binomial expressions are approximately the same. Note that the algorithm 4 can be computed in a distributed architecture since we can merge *PowerSeries* (see subroutine in algorithm 4). Although we compute approximately $(b-2)n$ more additions in a binomial expression, we show later that an intermediate result, a *PowerSeries(X,b)*, can be used to share computations, which can significantly compensate this additional cost.

- *Sharing PowerSeries*. We explain as follows a binomial transformation brings more possibilities to share computations. Assuming that in a query we need to compute the following m aggregation states $(s_1, ..., s_m)$ with $s_j(X) = \sum_{i=1}^{n} (x_i + a_j)^{b_j}, j \in (1, ..., m)$ where a_j is a constant and b_j is a positive integer and $b_j \geqslant 2$. According to the cost analysis, depicted in Table A.1, it requires $2mn$ additions and $n(b_1 + ... + b_m - m)$ multiplications. While, if $(s_1, ..., s_m)$ are transformed using binomial transformations, then it needs to compute m PSs (*PowerSeries*), that $PSj(X, b_j) = (\sum x_i, ..., \sum x_i^{b_j})$. It is trivial to see that we only need to compute $PS_{max}(X, b_{max})$ with $b_{max} = max(b_1, ..., b_m)$, which can be shared to each of the m PSs. If we do like this, then it requires $nb_{max} + b_1 + ... + b_m$ additions and $n(b_{max} - 1) + f(b_1) + ... + f(b_m)$ multiplications with $f(b) = \frac{(b+2)(b-1)}{2} + b$. Since every $b_j, j \in (1, .., m)$ is much smaller than n, then we can approximately avoid $n(b_1 + ... + b_m - m - b_{max} + 1)$ multiplications. The problem of whether we can reduce additions, or how many we can reduce, will depend on b_{max} and m. If $2m > b_{max}$, the reduced cost is significant since both additions and multiplications can be reduced. Now, assuming the m aggregation states are separately computed in m queries. Then, we can cache the results of earlier *PowerSeries* and reuse them for later ones. In the best case, where PS_{max} comes at first, then we do not need to launch the computation for the latter ones. Note that, the sharing approach does not have any constraints. Such that, an aggregate state s_j of the m ones can contain an arbitrary a_j and $\forall b_j \in \mathbb{Z}_{\geqslant 2}$.

We observe that the subroutine *PowerSeries* in algorithm 4 cannot be expressed by SQL syntax. Such that, we wrap it using a Spark or PostgreSQL UDAF interface, which is defined as $PS()$. If we identify an aggregation state s with an original expression, i.e., $s(X) = \sum_{i=1}^{n} (x_i + a)^b$, then we transform s into a binomial expression and use the function $PS()$ to compute a *PowerSeries* contained in the binomial expression of s.

Algorithm 4: Computing $BinomialExpression$

Input: X, a, b
Output: $s(X) = \sum_{k=0}^{b} \binom{b}{k} a^{b-k} (\sum_{i=1}^{n} (x_i)^k)$
Function $BinomialExpression(X, n, a, b)$
  var double $ps[b] \leftarrow PowerSeries(X, b)$;
  var int k;
  var double s;
  for k:= 1 *to* b **do**
    $s \leftarrow s + \binom{b}{k} \times a^{b-k} \times ps[k-1]$;
  end
  return $s \leftarrow s + a^b \times n$;
Function $PowerSeries(X, b)$
  var double $temp[b]$;
  var double $ps[b]$;
  var double x;
  var int l;
  while $X.hasNext()$ **do**
    $x \leftarrow X.next()$;
    $temp[0] \leftarrow x$;
    $ps[0] \leftarrow ps[0] + x$;
    for l:= 1 *to* $b-1$ **do**
      $temp[l] \leftarrow x \times temp[l-1]$; ▷ Computing $x_i^p, p \geqslant 2$
      $ps[l] \leftarrow ps[l] + temp[l]$;
    end
  end
  return ps;
Function $Merge(ps1, ps2)$ ▷ *Merging two power series*
  var int j;
  for i:= 0 *to* $b-1$ **do**
    $ps1[j] \leftarrow ps1[j] + ps2[j]$;
  end
  return $ps1$;

Bibliography

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[apaa] Aache HIVE. https://hive.apache.org.

[apab] Apache Flink. https://flink.apache.org.

[apac] Apache Hadoop. https://hadoop.apache.org.

[apad] Apache Spark. https://spark.apache.org/.

[AW04] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.

[BKS17] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, October 2017.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[CN05] Zhimin Chen and Vivek Narasayya. Efficient computation of multiple group by queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 263–274, New York, NY, USA, 2005. ACM.

[CNS99] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 155–166, New York, NY, USA, 1999. ACM.

[CNS00] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Algorithms for rewriting aggregate queries using views. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems for Advanced Applications: Current Issues in Databases and Information Systems*, ADBIS-DASFAA '00, pages 65–78, London, UK, UK, 2000. Springer-Verlag.

[CNS06] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst.*, 31(2):672–715, June 2006.

[Coh05] Sara Cohen. Containment of aggregate queries. *SIGMOD Rec.*, 34(1):77–85, March 2005.

[Coh06] Sara Cohen. User-defined aggregate functions: Bridging theory and practice. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 49–60, New York, NY, USA, 2006. ACM.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB '94*, pages 354–366, 1994.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimizing queries with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 167–182, London, UK, UK, 1996. Springer-Verlag.

[CTK+16] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. Cutty: Aggregate sharing for user-defined windows. pages 1201–1210, 10 2016.

[dat19] Implement the userdefinedaggregatefunction: Geometricmean. https://docs.databricks.com/spark/latest/spark-sql/udaf-scala.html, 2019.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 197–208, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150, San Francisco, CA, 2004.

[Dij] Edsger Dijkstra. The shunting-yard algorithm. https://en.wikipedia.org/wiki/Shunting-yard_algorithm.

[DRSN98] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 259–270, New York, NY, USA, 1998. ACM.

[FMS+10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Trans. Algorithms*, 6(4):66:1–66:19, September 2010.

[GCoS+97] Jim Gray, Surajit Chaudhuri, BoswOrthogonal Optimization of Subqueries, Adam Aggregation, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, Mar 1997.

[GDT+18] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. Moment-based quantile sketches for efficient high cardinality aggregation queries. *Proc. VLDB Endow.*, 11(11):1647–1660, July 2018.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 358–369, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[GL01] Jonathan Goldstein and Per-Ake Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 331–342, New York, NY, USA, 2001. ACM.

[GLJ01] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 571–581, New York, NY, USA, 2001. ACM.

[GMMP11] Michel Grabisch, Jean-Luc Marichal, Radko Mesiar, and Endre Pap. Aggregation function: Means. *Information Sciences*, 181(1):1–22, January 2011.

[GSZ11] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Proceedings of the 22Nd International Conference on Algorithms and Computation*, ISAAC'11, pages 374–383, Berlin, Heidelberg, 2011. Springer-Verlag.

[GT00] Stéphane Grumbach and Leonardo Tininini. On the content of materialized aggregate views. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, pages 47–57, New York, NY, USA, 2000. ACM.

[HGHS07] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. Sharing Aggregate Computation for Distributed Queries. In *Proc. 2007 ACM SIGMOD Int. Conf. Manag. Data*, SIGMOD '07, pages 485–496, New York, NY, USA, 2007. ACM.

[HMJJ00] H.Garcia-Molina, J.D.Ullman, and J.Widom. *Database System Implementation*. Prentice-Hall, New Jersey, 2000.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[HPK+13] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. 04 2013.

[Ita15] Telecom Italia. Telecommunications - sms, call, internet - mi, 2015.

[KBY17] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD '17*, pages 1717–1722, 2017.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD '06*, pages 623–634, 2006.

[LMS95] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '95, pages 95–104, New York, NY, USA, 1995. ACM.

[MGP09] Radko Mesiar Michel Grabisch, Jean-Luc Marichal and Endre Pap. *Aggregation Functions*. Cambridge University Press, Cambridge, 2009.

[MLT15] M.Jean-Luc and T.Bruno. Preassociative aggregation functions. *Fuzzy Sets and Systems*, 268:15–26, June 2015.

[MLT16] M.Jean-Luc and T.Bruno. Strongly barycentrically associative and preassociative functions. *Fuzzy Sets and Systems*, 437(1):181–193, May 2016.

[NP06] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB '06*, pages 1049–1058, 2006.

[onl] Jsqlparser. https://github.com/JSQLParser/JSqlParser.

[onl18] Cauchy's functional equation. https://en.wikipedia.org/wiki/Cauchy%27s_functional_equation, 2018.

[onl19a] Central moments. https://en.wikipedia.org/wiki/Central_moment, 2019.

[onl19b] Standardized moments. https://en.wikipedia.org/wiki/Standardized_moment, 2019.

[Par19] https://www.postgresql.org/docs/current/parallel-plans.html, 2019.

[RDBa] IBM DB2. https://www.ibm.com/analytics/db2.

[RDBb] Oracle. https://docs.oracle.com/.

[RDBc] PostgreSQL. https://www.postgresql.org/docs/.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 318–329, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[SF95] Michael Stillger and Johann-Christoph Freytag. An overview of cost-based optimization of queries with aggregates. *IEEE Data Eng. Bull.*, 18:10–18, 1995.

[Sma07] Christopher G. Small. *Functional Equations and How to Solve Them*. Springer-Verlag New York, 2007.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD '16*, pages 3–18, 2016.

[sta18] Moment-based quantile sketches for aggregations. https://github.com/stanford-futuredata/msketch, 2018.

[WWDI17] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *SIGMOD '17*, pages 557–572, New York, NY, USA, 2017. ACM.

[YbL95] Weipeng P. Yan and Per bike Larson. Eager aggregation and lazy aggregation. In *VLDB '95*, pages 345–357, 1995.

[YGI09] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP '09*, pages 247–260, New York, NY, USA, 2009. ACM.

[YL94] W. P. Yan and P. . Larson. Performing group-by before join /spl lsqb/query processing/spl rsqb/. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, pages 89–100, Feb 1994.

[ZT19] Chao Zhang and Farouk Toumani. Sharing computations for user-defined aggregate functions (technical report). 2019.

[ZTG17a] Chao Zhang, Farouk Toumani, and Emmanuel Gangler. Efficient computation of aggregate functions in large scale data processing frameworks. In *XLDB '17*, 2017.

[ZTG17b] Chao Zhang, Farouk Toumani, and Emmanuel Gangler. Symmetric and asymmetric aggregate functions in massively parallel computing. In *VLDB '17(PhD workshop)*, 2017.

[ZTG18] Chao Zhang, Farouk Toumani, and Emmanuel Gangler. Decomposing and sharing user-defined aggregation: from theory to practice. In *BDA '18*, 2018.