Decomposing and Sharing for User-defined Aggregation: from Theory to Practice

Chao Zhang, Farouk Toumani

▶ **To cite this version:**

Chao Zhang, Farouk Toumani. Decomposing and Sharing for User-defined Aggregation: from Theory to Practice. 2018. hal-01877088v1

HAL Id: hal-01877088

https://hal.science/hal-01877088v1

Preprint submitted on 19 Sep 2018 (v1), last revised 18 Oct 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decomposing and Sharing User-defined Aggregation: from Theory to Practice

ZHANG Chao
LIMOS et LPC, Université Clermont Auvergne
zhangch@isima.fr

Farouk Toumani
LIMOS, Université Clermont Auvergne
ftoumani@isima.fr

ABSTRACT

We study the problems of decomposing and sharing user-defined aggregate functions in distributed and parallel computing. Aggregation usually needs to satisfy the distributive property to compute in parallel, and to leverage optimization in multidimensional data analysis and conjunctive query with aggregation. However, this property is too restricted to allow more aggregation to benefit from these advantages. We propose for user-defined aggregation functions a formal framework to relax the previous condition, and we map this framework to the $\mathcal{MRC}$, an efficient computation model in MapReduce, to automatically generate efficient partial aggregation functions. Moreover, we identify the complete conditions for sharing the result of practical user-defined aggregation without scanning base data, and propose a hybrid solution, the symbolic index, pull-up rules, to optimize the sharing process.

KEYWORDS

MapReduce, Large-scale, user-defined aggregation, decomposing aggregation, sharing computation.

1 INTRODUCTION

The ability to summarize information, the intrinsic feature of aggregation, is drawing increasing attention for information analysis [9, 18]. Simultaneously under the progress of data explosive growth, processing aggregate function has to experience a transition to massively distributed and parallel framework, e.g. MapReduce, Spark, Flink etc. The inherent property of aggregation, taking several values as input and generating a single value based on certain criteria, requires a decomposing approach in order to be executed in a distributed architecture. Decomposing aggregation functions enables to compute partial aggregation which can then be merged together to obtain the final result. These partial aggregation results also require an efficient managing method for exhaustively reusing due to the fact that data scanning is usually costly in distributed and parallel computing. *How to efficiently decompose user-defined aggregation functions and exhaustively reuse their partial computation results* is a hard nut to crack.

Decomposition of aggregation function is a long-standing research problem that has been addressed in various fields. In a distributed computing framework like MapReduce, decomposability of aggregate function can push aggregation before the shuffle phase [5, 26]. This is usually called initial reduce, with which the size of data transmission on a network can be substantially reduced. For wireless sensor network, the need to reduce data transmission is more necessary because of the limitation of power supply [23]. In online analytical processing (OLAP), decomposability of aggregate function enables aggregation across multi-dimensions, such that aggregation queries can be executed on pre-computation results instead of base data to accelerate query answering [24]. An important point of query optimization in relational databases is to reduce input table size of join [16], and decomposable aggregation brings interests [6].

Previous works identify interesting properties for decomposing aggregation. A very relevant classification of aggregation functions, introduced in [18], is based on the size of sub-aggregation (i.e., partial aggregation). This classification distinguishes between distributive and algebraic aggregation, having sub-aggregate with fixed sizes, and holistic functions, where there is no constant bound on the storage size needed to describe a subaggregation. Some algebraic properties, such as associativity and commutativity, are identified as sufficient conditions for decomposing aggregation [5, 26]. Compared to these works, our work provides a generic framework to identify the decomposability of any symmetric aggregation and generate generic algorithms to process it in parallel.

On the other side of sharing aggregation computation, [8, 10, 15, 27] focus on aggregate functions with varying selection predicates and group-by attributes. In dynamic data processing, [14, 17, 20] concentrate on windowed aggregate queries with different ranges and slides. [25] proposes a framework to manage the partial aggregation results, and it has shown performance improvement compared to modern data analysis library e.g. Numpy. Previous works focus on optimizing queries with aggregation functions having different group attributes, predicates, and windows (range and slide), while we concentrate on sharing computation results for completely different aggregation functions without these constraints (aggregation simply runs on input dataset). And our solutions can be trivially extended to relational queries with aggregation functions by exploiting the contribution of previous works.

We focus on designing a generic framework that enables to efficiently process user-defined aggregation functions and exhaustively reuse their computation results. To achieve this goal, we firstly identify a computation model and an associated cost model for parallel algorithms. We consider in our work the MapReduce (*MR*) framework and we use the $\mathcal{MRC}$ [19] cost model to define *'efficient' MR* algorithms. Then we rest on the well-formed aggregation [6] as a generic framework for aggregation functions. This formal framework is mapped into the $\mathcal{MRC}$ model to generate a generic efficient MR algorithm for aggregation in section 4, noted by $MR(\alpha)$. Moreover, in section 5, we identify the widely used $\oplus$ functions in practice in the formal framework and analyze their effects on the efficiency of $MR(\alpha)$. On the side of sharing aggregation computation, at first, in section 6 we introduce the sharing strategy based on the formal framework. Then in section 7, we completely identify the sharing conditions for practical user-defined aggregation. In order to improve the sharing process, the symbolic index

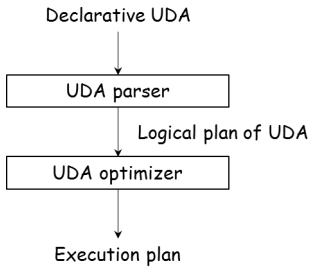

Figure 1: Outline of processing user-defined aggregation.

and pull-up rules are proposed to optimize the execution plan of aggregation.

This research is aiming at providing a systematic approach for processing user-defined aggregation to fit the increasing application of aggregation functions in different fields of data analysis. The general outline can be concluded in figure 1. In this paper, we concentrate on the formal framework of user-defined aggregation and the theoretical aspects of the optimizing phase.

2 *MRC* ALGORITHM

Several research works concentrate on the complexity of parallel algorithms. *MUD* [11] algorithm was proposed to transform a symmetric streaming algorithm to a parallel algorithm with decent bounds in terms of communication and space complexity, but without any bound on the time complexity. This disqualifies *MUD* as a possible candidate in our context. The trade-off between round numbers and reducer space has been analyzed in [12], and reducer memory is relaxed to an arbitrary number to solve complex problems e.g. prefix sums and multi-searching. In [4], the massively parallel communication model was proposed to analyze the trade-off between communication load and computation rounds for relational queries. *MRC* [19] is another popular framework that has been used to evaluate whether a MapeReduce algorithm is efficient. The constraints enforced by *MRC* w.r.t. to the total bits of input n can be summarized in the follows:

- machine space: $O(n^{1-\epsilon}),\ \epsilon > 0$;
- local computation time: $O(n^k)$, for some constant k;
- machine numbers: $O(n^{1-\epsilon}),\ \epsilon > 0$;
- computation round: $R = O(log^i(n))$.

The *MRC* model considers necessary parameters for parallel computing, communication time, computation space and computing time, and makes more realistic assumptions. Hence, a MapReduce algorithm satisfying these constraints is considered as an efficient parallel algorithm and will be called hereafter an *MRC* algorithm.

3 FORMAL FRAMEWORK

In this section, first of all, we formally define aggregation function. Then the symmetric (commutative) aggregation functions are introduced. Finally, we provide the formal framework for symmetric aggregation which is going to be used throughout this paper.

Definition 3.1. **(Aggregation function)** Let I be a domain (i.e., a set of infinite number of values). An aggregation function α over I is a function: $\bigcup_{l\in\mathbb{N}_{>1}} I^l \rightarrow I$.

According to this definition, corresponding to the notion of extended aggregation function in [13], an aggregation operates on a list of values to compute a single value as a result.

Definition 3.2. **(Symmetric aggregation function)** Let I be a domain. An aggregation function α is symmetric iff $\alpha(X) = \alpha(\sigma(X))$ for any $X \in I^l, l \in \mathbb{N}_{>1}$ and any permutation σ, where $\sigma(X) = (x_{\sigma(1)}, ..., x_{\sigma(l)})$.

Symmetric aggregation does not depend on the order of input data, therefore we consider the input of a symmetric aggregation is a *multiset* instead of an ordered list. For a given domain I, by noting $\{\{I\}\}$ as the set of all nonempty finite multisets of elements of I, a symmetric aggregation function is a function: $\{\{I\}\} \rightarrow I$.

To define the generic aggregation framework, we use the notion of well-formed aggregation in [6] as the *canonical form*.

Definition 3.3. (**Canonical form of symmetric aggregation function**) Let α be a symmetric aggregation function defined over a domain I and let D_i be a domain, called an intermediate domain. A canonical form of α using an intermediate domain D_i is a triple $(F, \oplus, T)$,

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

such that for all $\{d_1, ..., d_l\} \in \{\{I\}\}$, $\alpha(\{d_1, ..., d_l\}) = T(F(d_1) \oplus \ldots \oplus F(d_l))$.

F is a tuple at a time function operating on single values of I. The binary operation $\oplus$ accumulates results of F and hence plays the role of an accumulator, and T operates on the accumulated results of $\oplus$ to finalize the computation of α. For instance, the aggregation $average(X)$ with $X \in \{\{I\}\}$ can be expressed in the following canonical form, called hereafter *canaverage* form:

- $F(d) = (d, 1), \forall d \in X$;
- $(d, k) \oplus (d', k') = (d + d', k + k')$;
- $T((d, l)) = \dfrac{d}{l}$.

More examples are illustratd in table 6 with main SQL built-in aggregation functions available on some commercial and open-source DBMSs (Microsoft SQL Server [2],IBM DB2 [1] and PostgreSQL [3]).

We make the following two observations: *(i)* a canonical form of an aggregation function is not unique, i.e., for the same function α, several canonical forms may exist, and *(ii)* an aggregation function α can always be expressed in a canonical form: taking F as the identity function, $\oplus$ as the multiset union and T as α itself. This latter form is called hereafter the *naive canonical form* of α.

4 EFFICIENT DECOMPOSING AGGREGATION

In this section, we firstly decompose aggregation functions using the formal framework, then we illustrate the deficiency of the formal framework, lacking the consideration of computing efficiency. Therefore, we map the formal framework into the $\mathcal{MRC}$ algorithm

to generate the efficient decomposing framework. After this, we show how the formal framework deals with several algebraic properties of aggregation functions which are commonly used in the decomposing literature, from this we can see that the formal framework is more generic than the previous algebraic properties.

4.1 Mapping symmetric aggregation into $\mathcal{MRC}$ algorithm

Let $(F, \oplus, T)$ be a canonical form of an aggregation function α. The associative and commutative property of $\oplus$ can be exploited to derive a MapReduce implementation of α: processing F and $\oplus$ at mapper, $\oplus$ at combiner, and $\oplus$ and T at reducer. Table 1 depicts the corresponding generic MapReduce algorithm, $MR(\alpha)$, to compute $\alpha(X)$ with $X \in \{\{I\}\}$, where mapper input is a submultiset $X_i \subseteq X$, the output of the mapper i is denoted by O_i, and the symbol $\sum_{\oplus}$ denotes the summation using $\oplus$.

Hence, every symmetric 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 $MR(canaverage)$, derived from the *canaverage* form of the *average* function is a $\mathcal{MRC}$ algorithm while the algorithm $MR(naiveaverage)$, derived from the naive 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 output of mappers is equal to n (the length of the input), and if, in the worst case, all the mapper outputs are sent to only one reducer it will need a space equal to n. However, $\mathcal{MRC}$ model requires that a reducer uses a sublinear space in n.

Therefore, we address in the sequel the following question: given an aggregation function α expressed in a canonical form, *when the generated $MR(\alpha)$ algorithm is efficient?* In other words, we are interested in characterizing under which conditions one can ensure that $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

In order to have conditions to make $MR(\alpha)$ be a $\mathcal{MRC}$ algorithm, it is necessary to reason about partial aggregation output length which is related to the bounded space of reducer in $\mathcal{MRC}$. Moreover, the extremely worst (EW) case in MapReduce is that all mapper outputs are sent to one reducer, and if the condition of partial aggregation result length can be satisfied in EW case, then the case of a general number of reducers can also be ensured. Therefore, our analysis is built on the EW case.

In $\mathcal{MRC}$ model, machine number P and machine space M are both restricted to $O(n^{1-\epsilon})$. However, this may bring a restricted condition that the bound of P and M must be same or same order of magnitude, which limits the feasibility under EW case of computing even trivial aggregation e.g. count(), because under EW case $P \cdot |O_i| = M$ where $|O_i|$ is the length of a mapper output O_i in bits, if P and M are same or same order of magnitude, $|O_i|$ can only be $O(1)$. Therefore, we assume machine number: $P = O(n^{1-\epsilon_1}), \epsilon_1 > 0$ and machine space: $M = O(n^{1-\epsilon_2}), \epsilon_2 > 0$, and both of them are still under the $\mathcal{MRC}$ sublinear requirement. In the following, we call $\mathcal{MRC}$ **environment** $E = (X, \epsilon_1, \epsilon_2)$ for an input $X = \{< k_j; v_j >, \text{for } j \in [1, l]\}$ of $n = \sum_{j=1}^{l}(|k_j| + |v_j|)$ bits, and a cluster of $P = O(n^{1-\epsilon_1}), \epsilon_1 > 0$ machines, and each of them having $M = O(n^{1-\epsilon_2}), \epsilon_2 > 0$ space.

Table 1: $MR(\alpha)$: a generic MR algorithm for symmetric aggregation.

MapReduce phase	operation
mapper	$\sum_{\oplus, d_j \in X_i} F(d_j)$
combiner	$\oplus$
reducer	$T(\sum_{\oplus, i} O_i)$

Proposition 4.1. *Given an MapReduce program $\mathcal{A}$ with the $\mathcal{MRC}$ computing environment, then under the EW case, $\mathcal{A}$ can be computed in the $\mathcal{MRC}$ environment $E = (X, \epsilon_1, \epsilon_2)$ iff for each mapper output O_i, $|O_i| = O(n^{1-\zeta}), \zeta \geqslant max(\epsilon_2,\ 1 - \epsilon_1 + \epsilon_2)$.*

Based on proposition 4.1, we propose the following reducible property to categorize when a symmetric aggregation has a $\mathcal{MRC}$ algorithm.

Definition 4.2. (**Reducible symmetric aggregation functions**) Let α be a symmetric aggregation given in a canonical form $(F, \oplus, T)$ with the $\mathcal{MRC}$ environment $E = (X, \epsilon_1, \epsilon_2)$, α is reducible if it satisfies the following two conditions:

- $F, \oplus$ and T operate in time polynomial in n;
- $|\sum_{\oplus, d_j \in X_i} F(d_j)| = O(n^{1-\zeta}), \zeta \geqslant max(\epsilon_2,\ 1 - \epsilon_1 + \epsilon_2)$, where X_i is a subset of X with the size of $O(n^{1-\epsilon_2})$ in bits.

The second condition requires that the output length of partial aggregation is bounded by a precisely sublinear space in n. This ensures that the length of partial aggregation is smaller enough to make the underlying computation efficient. It is noteworthy that this constraint is more general than the $\mathcal{MRC}$ parallelizable property in [19].

It is trivial to see that the formal framework is a one-round *MR* algorithm, then with reducible property we can conclude in the follows the $MR(\alpha)$ for a reducible α is a $\mathcal{MRC}$ algorithm, which means $MR(\alpha)$ can be efficiently computed in *MR* paradigm.

$$MR(\alpha), \alpha \text{ is reducible} \Leftrightarrow \mathcal{MRC} \text{ algorithm}. \quad (1)$$

4.2 Deriving $\mathcal{MRC}$ algorithm from algebraic properties

In this subsection, we investigate different algebraic properties of aggregation functions leading to a non-naive canonical form. If an aggregation α is in one of the following classes and α is also reducible, then the corresponding $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

Associative aggregation. An aggregate function α is *associative* [13] if for any multiset $X = X_1 \cup X_2$, $\alpha(X) = \alpha(\alpha(X_1), \alpha(X_2))$. Associative and symmetric aggregation function can be transformed in a canonical form $(F, \oplus, T)$ defined as follows:

$$F = \alpha,\ \oplus = \alpha,\ T = id. \quad (2)$$

where id denotes the identity function. If α is reducible, then $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

Distributive aggregation. An aggregation α is *distributive* [18] if there exists a combining function C such that $\alpha(X) = C(\alpha(X_1), \alpha(X_2))$.

Distributive and symmetric aggregation can be rewritten in the canonical symmetric aggregation framework $(F, \oplus, T)$:

$$F = \alpha,\ \oplus = C,\ T = id. \tag{3}$$

Similarly, if α is reducible then the corresponding $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

Commutative semigroup aggregation. Another kind of aggregate function having the same behavior as symmetric and distributive aggregation is the commutative semigroup aggregate function [7]. An aggregation α is in this class if there exists a commutative semigroup $(H, \otimes)$, such that $\alpha(X) = \otimes_{x_i \in X} \alpha(x_i)$. The corresponding canonical aggregation $(F, \oplus, T)$ is illustrated as following:

$$F = \alpha,\ \oplus = \otimes,\ T = id. \tag{4}$$

If α is reducible then $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm.

Preassociative and symmetric aggregation A more general property than commutative semi-group aggregation is symmetric and preassociative aggregate function. An aggregation α is *preassociative* [21] if it satisfies $\alpha(Y) = \alpha(Y') \implies \alpha(XYZ) = \alpha(XY'Z)$. According to [21], some symmetric and preassociative aggregation functions (unarily quasi-range-idempotent and continuous) can be constructed as $\alpha(\mathbf{X}) = \psi\left(\sum_{i=1}^{n} \varphi(x_i)\right), n \geq 1$, where ψ and φ are continuous and strictly monotonic unary function. A canonical form $(F, \oplus, T)$ for this kind of preassociative aggregation can be defined as following:

$$F = \varphi,\ \oplus = +,\ T = \psi. \tag{5}$$

The corresponding $MR(\alpha)$ is also a $\mathcal{MRC}$ algorithm, if α is reducible. For instance, $\alpha(X) = \sum_{i=1}^{n} 2x_i$, where $\psi = id$ and $\varphi(x_i) = 2x_i$.

Quasi-arithmetic mean. An aggregate function α is barycentrically associative [22] if it satisfies $\alpha(XYZ) = \alpha(X\alpha(Y)^{|Y|}Z)$, where $|Y|$ denotes the number of elements contained in multiset Y and $\alpha(Y)^{|Y|}$ denotes $|Y|$ occurrences of $\alpha(Y)$. A well-known class of symmetric and barycentrically associative aggregation is quasi-arithmetic mean: $\alpha(\mathbf{X}) = f^{-quasi}\left(\frac{1}{l}\sum_{i=1}^{l} f(x_i)\right), l \geq 1$, 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:

$$F = (f, 1),\ \oplus = (+, +),\ T = f^{-1}(\frac{\sum_{i=1}^{l} f(x_i)}{l}). \tag{6}$$

The corresponding $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm, if α is reducible.

5 PRACTICAL DECOMPOSING

In this section, we firstly identify two widely used $\oplus$ functions, which are adequately generic to construct user-defined aggregation based on the analysis of table 6 (see at the last page). Then we analyze the relationship between these two $\oplus$ functions and the reducible property.

5.1 Efficient $\oplus$ function

According to the canonical forms of most common used aggregation (see table 6), the $\oplus$ function in the formal framework can be set union, addition, and multiplication, or a tuple of the combination of them. It is trivial to see that set union is not an efficient $\oplus$ function, because of materializing all data from slave nodes to master nodes. In practice, for aggregation functions which can only have set union as the $\oplus$ function, e.g. median, approximated algorithms are usually used to compute them, which is out the scope of this paper.

The reducible property gives the necessary and sufficient condition to identify when an arbitrary aggregation in the canonical form is a $\mathcal{MRC}$ algorithm. From definition 4.2, we observe that partial aggregation output length is bounded, which depends on input value length $|v|$ in bits and the increasing bits by using $\oplus$ to accumulate values.

When it comes to practical computing, there is always a trade-off between the length of the result and the computation precision (unbounded-length result). For instance, in Java, the primitive data type double has a fixed length of 64-bit, but the precision of computation result using double is out of control. While, the BigDecimal provides arbitrary precision arithmetic, but the result has an unbounded length depending on precision. For the bounded-length data type, the $MR(\alpha)$ with $\oplus = +$ or $\oplus = \times$ is always a $\mathcal{MRC}$ algorithm, because there is no increase on the size of accumulating values, in other words, the size of the mapper output is always the bounded number of bits, which could be $O(1)$. We analyze $MR(\alpha)$ with the unbounded-length data type in the follows, which is for the cases of arbitrary precision computing and exact computing with unlimited precision.

5.1.1 $\oplus = +$. We show in the following theorem, under the case of unbounded data type, when a $MR(\alpha)$ with $\oplus = +$ in the $\mathcal{MRC}$ environment is a $\mathcal{MRC}$ algorithm.

Theorem 5.1. *Let α be a symmetric aggregation in a canonical form with $\oplus = +$ and F, T operating in time polynomial in the length of input, and $MR(\alpha)$ be the corresponding generic algorithm with the $\mathcal{MRC}$ environment $E = (X, \epsilon_1, \epsilon_2)$, when $|F(v_i) + F(v_j)|, \forall v_i, v_j \in X$ is unbounded, $MR(\alpha)$ is a $\mathcal{MRC}$ algorithm iff $|avg(F(v_i))| = O(n^{1-\zeta}), \zeta > max(\epsilon_2,\ 1 - \epsilon_1 + \epsilon_2)$, where $avg(F(v_i))$ is the average value of the $\cup_{v_i \in X} F(v_i)$.*

In fact, the bound of $|avg(F(v_i))|$ is quite generic in practice. With respect to the $\mathcal{MRC}$ environment, the bound $O(n^{1-\zeta})$ equals to $\frac{M}{P}$, which is the machine space in bits divided by the machine number. And this can be quite large in a practical environment. Therefore, under most reasonable cases, $MR(\alpha)$ with $\oplus = +$ is a $\mathcal{MRC}$ algorithm.

5.1.2 $\oplus = \times$. Generally, in the setting of processing aggregation in distributed systems, if an aggregation function is associative and commutative then the corresponding partial aggregation can be efficiently processed. We show the following counterexample of $\oplus = \times$, that this widespread practice is in fact not correct with consideration of $\mathcal{MRC}$ cost model in the case of the unbounded-length data type.

Given the computation of the product aggregation $\alpha(X) = \prod_{v_j \in X} v_j$, let $X = \{v_j, \forall j \in [1, l]\}$ where v_j is a binary string. W.o.l.o.g., consider $v_j > 0$. The total length of the input for α is $\sum_{v_j \in X} |v_j| = \sum_{v_j \in X} log(v_j)$ bits. 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**. W.o.l.o.g., we ignore

Table 2: Computation time and total partial result size for $\sum x$ and $\prod x$ with unlimited precision.

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

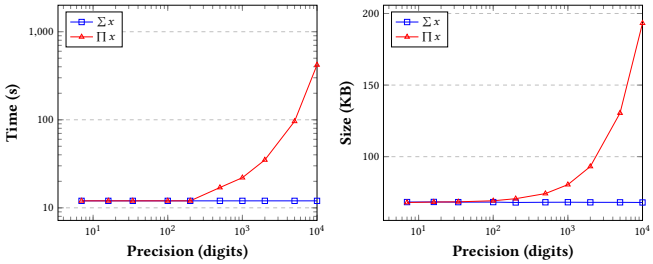

Figure 2: Computation time and total partial result size for $\sum x$ and $\prod x$ with different significant digits.

the **Combiner** phase since it does not impact our reasoning. The computation results of a partial aggregation $\alpha(X_i)$ is $\prod_{v_j \in X_i} v_j$, of which encoding requires $log(\prod_{v_j \in X_i} v_j) = \sum_{v_j \in X_i} log(v_j)$ bits. In the worst case of MapReduce computing, all the results of the mappers (the case of one mapper at each machine) are sent to a unique reducer. Hence the reducer will need a space equal to $\sum_{X_i \in X}(\sum_{v_j \in X_i} log(v_j))$ bits, which is indeed $\sum_{v_j \in X} log(v_j)$. In another sense, the computation of product contains shuffling all input data on all mappers to one reducer.

5.1.3 Experiments. Based on the above analysis, we make a simple experiment of computing $\sum x$ and $\prod x$ using arbitrary precision. The dataset we use is the store_sales table from TPC-DS generated by scale 10, and the size of the table is 3.72GB. We program the following two queries in the way of Spark RDD with different predefined precision and unlimited precision and run them on a spark cluster containing one master node and six slave nodes. The experiment results for the predefined precision case is illustrated in figure 2, and table 2 shows the computation time and total partial result size in the case of unlimited precision (exact computing).

```
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 can see both $\sum x$ and $\prod x$ can be efficiently processed with less digits in precision (smaller length of results). When using more digits and even unlimited digits, $\sum x$ stays on the same performance, while $\prod x$ will dramatically increase in terms of computation time and result size.

5.1.4 The scope of $\oplus$. The $\oplus$ function can be a binary function or a tuple of binary function (for the case of not only one partial aggregation). Then, the scope of the $\oplus$ function is $\oplus = \cup_{i=1}^{n} \oplus_i$, $\oplus_i \in (+, \times)$, $i \in (1, ..., n), n \in \mathbb{N}_{>0}$.

6 SHARING AGGREGATION

In this section, we target the problem of how to reuse aggregation results to avoid data access. Let α and β be two aggregation functions and X be an input set, and we try to cache the computation results of $\alpha(X)$ and reuse them to compute $\beta(X)$. We build aggregation pipelines based on the formal framework for α and β in figure 3, which are compared in the follows based on the size of data scanning and the scope of reusing.

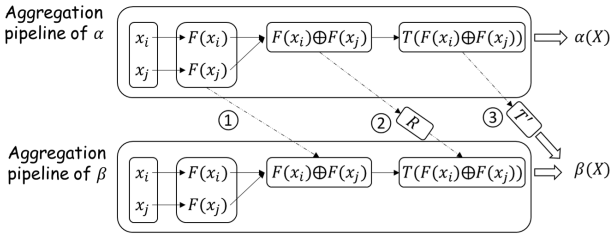

Figure 3: Sharing aggregation pipeline.

The first one is caching the results of translating function. However, this is not interesting because the size of its output equals to the one of the original dataset such that it still needs a same-scale data scanning.

The second one is caching the results of $\oplus$ function. Let PA_α be the total results of $\oplus$ function in α, which can be one or a tuple of values, and T_α be the terminating function of α, then $\alpha(X) = T_\alpha \circ PA_\alpha$. Let R be one or a tuple of unary functions, then if possible β can be computed as $\beta(X) = T_\beta \circ R \circ PA_\alpha$. R contains only unary functions, then the results of R only depend on the input PA_α instead of X. In fact, PA_α is a tuple containing aggregation results, and the size of PA_α is much smaller then the size of X.

The third one is caching the results of $\alpha(X)$, which is a single value because α is an aggregation. Let T' be a unary function, then if possible β can be computed as $\beta(X) = T' \circ \alpha(X)$. The difference between the second and third caching choice is $\alpha(X)$ can always be inferred from PA_α, but if we have $\alpha(X)$ we cannot always recover PA_α from it, because it requires the inverse function of T_α exists or T_α is a uni-variate injection. This is a quite restricted condition because the terminating function is usually not a univariate function, e.g. the one of average, or an injection, e.g. the one of $(\sum x)^2$. This difference determines the reusing scopes of them. If $\alpha(X)$ can be reused then PA_α must also be possible, but in the opposite way, it may not be possible. Such that, the reusing scope of the second caching choice always contain the scope of the third one.

Therefore, we choose the second caching and reusing choice. It is noteworthy that the unary functions in R may not be the identity function, such that α and β are not necessarily identical.

6.1 Partial aggregation state

First of all, we illustrate the notion of partial aggregation based on the formal framework. Given an aggregation function $\alpha : \cup_{l \in \mathbb{N}_{>1}} I^l \rightarrow I$ with the formal framework $(F, \oplus, T)$, then for any input $X \subseteq I$ the *partial aggregation* is

$$\sum_{\oplus, x_j \in X} F(x_j) = (\sum_{\oplus_1, x_j \in X} f_1(x_j), ..., \sum_{\oplus_m, x_j \in X} f_m(x_j)), m \in \mathbb{N}_{>0},$$

where $f_k(x), k \in (1, ..., m)$ is a unary function[1]. Partial aggregation can be trivially computed by processing it individually on all subsets of X and then combining these sub-results by $\oplus$. In this setting, the total result is cached and reused instead of sub-results in order to avoid data transferring on networks. As we can see, partial aggregation is a sequence containing aggregate functions. For instance, the partial aggregation of average is $(\sum_{x_j \in X} x_j, \sum_{x_j \in X} 1)$.

Moreover, we define a notion having a smaller granularity than partial aggregation, which is named hereafter *partial aggregation state*. A partial aggregation state is just an element in partial aggregation. In the above example, the partial aggregation of α contains m partial aggregation states

$$\sum_{\oplus_k, x_j \in X} f_k(x_j), k \in (1, ..., m).$$

Partial aggregation states can provide a more generic reusing method. We illustrate this benefit by using the previous sharing example between α and β. Assuming that γ is also an aggregation function, if $\sum_{\oplus_\gamma} F_\gamma \in (\sum_{\oplus_\alpha} F_\alpha, \sum_{\oplus_\alpha} F_\alpha)$, then the partial aggregation states of α and β can be reused for γ. Therefore, we formally define the sharing computation on aggregation state in the follows.

Definition 6.1. (**Sharing partial aggregation states**) Let $s = \sum_{\oplus} f(x)$ and $s' = \sum_{\oplus'_2} f'(x)$ be two partial aggregation states, then s' shares the result of s iff there exists a unary function r such that

$$s' = r \circ s.$$

Note that r is a unary function, which means it only depends on s instead of base data. Moreover, in the above sharing equation, if $s = s'$, then r is the identity function. If $s \neq s'$, by finding r, the results of s can also be reused for s'.

6.2 Non-trivial property: derivable set

Following the sharing definition 6.1, we consider the following decision problem, *given two aggregation states s and s', and an arbitrary input set, whether there exists a unary function r, such that the computation results of s can be reused for s', $s' = r \circ s$.* In order to answer this decision problem, we firstly propose an abstract structure on partial aggregation states named derivable set. Then, we have that for an aggregation state s, the derivable set of s is a non-trivial and semantic property. Finally, we conclude this decision problem is undecidable, according to the Rice's theorem.

Definition 6.2. (**Derivable set**) Let s and s' be two partial aggregation states with an arbitrary input set, then s' belongs to the derivable set $D(s)$, iff there exists a unary function r for $s' = r \circ s$,

$$D(s) = \{s' | s' = r \circ s\}.$$

Every derivable set contains at least one element, which is the case r can only be the identity function. Also, it is trivial to see the problem of deciding whether s' can share s is identical to decide whether $s' \in D(s)$.

Theorem 6.3. *Given any two partial aggregation states s and s', whether $s' \in D(s)$ is undecidable.*

[1]In this paper, the term unary function is used to denote a function taking only one argument as input, e.g. an element in an input set.

Table 3: Primitive types of unary functions.

Function name	Formula
Constant function	$f(x) = a, x \in \mathbb{R}, a \in \mathbb{R}$.
Identity function	$f(x) = x, x \in \mathbb{R}$.
Linear function	$f(x) = ax, x \in \mathbb{R}, a \in \mathbb{R}_{\neq 0}$.
Logarithm	$f(x) = log_a\|x\|, x \in \mathbb{R}_{\neq 0}, a \in \mathbb{R}_{>0,\neq 1}$.
Power function	$f(x) = \begin{cases} x^a, & x \in \mathbb{R}_{>0}, a \in \mathbb{R}_{\neq 0}. \\ x^a, & x \in \mathbb{R}_{<0}, a \in \mathbb{Z}_{\neq 0}. \end{cases}$
Exponential function	$f(x) = a^x, x \in \mathbb{R}, a \in \mathbb{R}_{>0}$.

7 PRACTICAL SHARING

In this section, we firstly present the practical scope of aggregation states, where the decision problem is decidable. After this, a naive algorithm is proposed to share aggregation state computation. Finally, in order to reduce time complexity, we propose a hybrid method containing a symbolic index and several pull-up rules to better share aggregation states.

7.1 Practical framework

Let $s = \sum_{\oplus} f(x)$ be a partial aggregation state. We already illustrate the practical scope of the $\oplus$ function in aggregation states, that $\oplus \in (+, \times)$. We introduce the scope of the unary function f by the following observation.

Observation We firstly observe that combining two unary functions by a functional composition or a bianry function is also a unary function. For instance, given $f_1(x) = x^b, b \neq 0$ and $f_2(x) = ax, a \neq 0$, both $f_2 \circ f_1(x) = ax^b$ and $f_2(x) + f_1(x) = x^b + ax$ are unary functions. Moreover, we also observe from table 6 (see in the appendix) that there are several primitive types of unary functions used in practice, e.g linear functions and power functions, and the other ones are built on top of these primitive functions.

Based on this observation, we identify several primitive types of unary and binary functions, which we summarize as the primitive unary function set (PU) and the primitive binary function set (PB).

- PU : let p be a unary function, then $p \in PU$ if p is one of *constant functions, the identity function, linear functions, power functions, logarithmic functions, and exponential functions* (see table 3 for the primitive functions).
- PB : let $\odot$ be a binary function, then $\odot \in PB$ if $\odot$ is one of arithmetic *addition, subtraction, multiplication and division.*

According to our observation, two types of unary functions can be built on top of PU using the functional composition $\circ$ and the binary functions in PB, which we summarize as the composing unary function set PU° and the concatenating unary function set $PU^\odot$.

- PU°: let g be a unary function, then $g \in PU^\circ$ if $g = p_l \circ ... \circ p_1$, $p_j \in PU$, $j \in (1, ..., l)$, $l \in \mathbb{N}_{>0}$, where p_1 is the first primitive function of g, and p_l is the last one, and the length of g is l, denoted as $|g| = l$.
- $PU^\odot$: let r be a unary function, then $r \in PU^\odot$ if $r = g_k \odot_{k-1} ... \odot_1 g_1$, $g_j \in PU^\circ$, $j \in (1, ..., k), \odot_z \in PB, z \in (1, ..., k-1)$, $k \in \mathbb{N}_{>1}$.

We illustrated some properties for the elements in PU° and $PU^\odot$, which will be used later to identify sharing conditions. For $f \in PU^\circ$

and f is not a constant function, we can have either f is an injection or f satisfying $f(x) = f(-x)$, because all primitive functions p besides constant functions are either injection or $p(x) = p(-x)$. Moreover, most unary functions in $PU^{\odot}$ are not injections because of the binary functions $\odot$. However, some specific elements in $PU^{\odot}$ can be transformed to PU° and become injections, e.g. $a_1x^b + a_2x^b = (a_1 + a_2)x^b = (a_1 + a_2)x \circ x^b$. These transformations require very specific conditions and can be trivially recognized by exhaustive analysis, such that we implement them but do not cover the details in the following part where we consider they have already been transformed.

With the scope of $\oplus$ and f, we present the following scope of user-defined aggregation functions covered by our decomposing and sharing framework.

Definition 7.1. Let α be an aggregation function with the formal framework $(F^{\alpha}, \oplus^{\alpha}, T^{\alpha})$ and X be the input of α, then α is in the *universe* U if

- $X \subseteq \mathbb{R}^l, l \in \mathbb{N}_{>1}$;
- $F^{\alpha} = (f_1, \dots f_m),\ f_i \in PU^{\odot} \cup PU^{\circ},\ i \in (1, \dots, m), m \in \mathbb{N}_{>0}$;
- $\oplus^{\alpha} = (\oplus_1, \dots, \oplus_m),\ \oplus_i \in (+, \times),\ i \in (1, \dots, m), m \in \mathbb{N}_{>0}$.

T is released to be any function, because T is the one computed at master node locally, such that no matter in decomposing phase or in sharing phase, T does not need special consideration.

7.2 Sharing practical aggregation states

In this subsection, we mainly propose complete conditions for sharing aggregation states having unary functions in PU°, and two transforming rules for those having ones in $PU^{\odot}$. Before exhaustively analyzing unary functions in PU° or $PU^{\odot}$, we show in the following theorem that the sharing possibility is related to whether unary functions are injections, which will be used to analysis PU° and $PU^{\odot}$.

Theorem 7.2. *Let $s_1 = \sum_{\oplus_1} f_1(x)$ and $s_2 = \sum_{\oplus 2} f_2(x)$ be two partial aggregation states, then there exists non-identity unary function r_{12} for $s_1 = r_{12} \circ s_2$ only under the following two situations,*

1. *f_2 is an injection;*
2. *neither f_1 nor f_2 is an injection.*

7.2.1 $f \in PU^{\circ}$. If $f \in PU^{\circ}$, we can have either f_2 is an injection or f_2 satisfying $f_2(x) = f_2(-x)$. We firstly illustrate complete conditions for the former case, and then transforming the latter case into the former cases by proposing the $sign(x)$ function. Note that, if f_1 is a constant function, it is trivial to compute it by using the aggregation results of *count*. Such that, we let f_1 be a non-constant function.

Case 1: $f_2, f_2 \in PU^{\circ}$ is an injection. We analyze the sharing conditions between s_1 and s_2 in an exhaustive way with respect to the practical scope of $\oplus$ function. Based on the two possible choices of $\oplus$ functions in aggregation states, there exists four different combinations for $\oplus_1$ and $\oplus_2$ in s_1 and s_2, that (1) $\oplus_1 = +, \oplus_2 = +$, (2) $\oplus_1 = +, \oplus_2 = \times$, (3) $\oplus_1 = \times, \oplus_2 = +$, (4) $\oplus_1 = \times, \oplus_2 = \times$. We present the sharing conditions for these four cases in the following four theorems.

Theorem 7.3. *Let $s_1 = \sum f_1(x)$ and $s_2 = \sum f_2(x)$ be two partial aggregation states having inputs of a set of real numbers, where $f_1(x)$ is a continuous non-constant unary function and $f_2(x)$ is a continuous injection, then there exists a non-identity unary function r_{12} for $s_1 = r_{12} \circ s_2$, if and only if $f_1 \circ f_2^{-1}(x) = r_{12}(x) = ax, x \in \mathbb{R}, a \in \mathbb{R}_{\neq 0}$.*

Theorem 7.4. *Let $s_1 = \sum f_1(x)$ and $s_2 = \prod f_2(x)$ be two partial aggregation states having inputs of a set of real numbers, where $f_1(x)$ is a continuous non-constant unary function and f_2 is a continuous injection, then there exists a non-identity unary function $r_{12}(x)$ for $s_1 = r_{12} \circ s_2$, if and only if $f_1 \circ f_2^{-1}(x) = r_{12}(x) = a(log_b|x|), x \in \mathbb{R}_{\neq 0}, b \in \mathbb{R}_{>0,\neq 1}, a \in \mathbb{R}_{\neq 0}$.*

Theorem 7.5. *Let $s_1 = \prod f_1(x)$ and $s_2 = \sum f_2(x)$ be two partial aggregation states having inputs of a set of real numbers, where f_1 is a continuous non-constant unary function and f_2 is a continuous injection, then there exists a non-identity unary function r_{12} for $s_1 = r_{12} \circ s_2$, if and only if $f_1 \circ f_2^{-1}(x) = r_{12}(x) = b^{ax}, x \in \mathbb{R}, b \in \mathbb{R}_{>0,\neq 1}, a \in \mathbb{R}_{\neq 0}$.*

Theorem 7.6. *Let $s_1 = \prod f_1(x)$ and $s_2 = \prod f_2(x)$ be two partial aggregation states having inputs of a set of real numbers, where f_1 is a continuous non-constant unary function and f_2 is a continuous injection, then there exists a non-identity unary function r_{12} for $s_1 = r_{12} \circ s_2$, if and only if $f_1 \circ f_2^{-1}(x) = r(x) = x^a,\ x \in R_{>0}, a \in \mathbb{R}$, or $f_1 \circ f_2^{-1}(x) = r(x) = x^a,\ x \in R_{<0}, a \in \mathbb{Z}_{\neq 0}$,*

Case 2: $f_2, f_2 \in PU^{\circ}$ satisfies $f_2(x) = f_2(-x)$. In this case, f_2 is not an injection, such that the above theorems can not be directly applied. However, f_2 satisfies the property $f_2(x) = f_2(-x)$, which can be used to transform f_2 to be an injection. In order to accomplish this, we propose a function called $sign(x)$ defined in the following. Then we let $u = sign(x) \times x$. Then, it is trivial to see that $u \geqslant 0$. Without loss of generality, we take $u > 0$ because some functions are not defined on 0.

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

With the $sign(x)$ function and the variable u, in order to check the sharing possibility of $s_1 = \sum_{\oplus_1} f_1(x)$ and $s_2 = \sum_{\oplus_2} f_2(x)$, the following two ones are verified $s_1' = \sum_{\oplus_1} f_1(u)$ and $s_2' = \sum_{\oplus_2} f_2(u)$. In this case, we indeed have $f_2(u)$ is an injection, such that the above four theorems can be directly applied.

After transforming the variable from x to u, we identify the relation between aggregation states on x and aggregation states on u. It is trivial to see that $s_2 = s_2'$ because of $f_2(x) = f_2(u), u = sign(x)$. Then, for the cases f_1 also satisfying $f_1(x) = f_1(-x)$, we also have $s_1 = s_1'$, such that the sharing problem of s_1' and s_2' is equivalent to the one of s_1 and s_2.

Finally the only left case is $f_1(x)$ is an injection, and $f_2(x)$ satisfies $f_2(x) = f_2(-x)$. According to theorem 7.2, there does not exist a unary function r_{12} for $s_1 = r_{12} \circ s_2$. However, a precomputation can be proceeded to make r_{12} to be an 'unary' function. Similarly, we mainly analyze the relation between s_1 and s_1' because we already have $s_2 = s_2'$. We firstly separate the input data sets into positive and negative parts, then we compensate $\sum_{\oplus_1} f_1(-x^-)$ where x^- is $\forall x < 0$. Then we have

$$s_1 = s_1' \oplus_1 C,$$

where $C = (\sum_{\oplus_1} f_1(x^-) \oplus_1^{-1} \sum_{\oplus_1} f_1(-x^-))$, and $\oplus_1^{-1}$ is the inverse function $\oplus_1$, e.g. $-$ is the inverse function for $+$, or $/$ is the one for and $\times$. Such that, $s_1' = s_1 \oplus_1^{-1} C$. Therefore, if there exists r_{12}' for $s_1' = r_{12}' \circ s_2'$, then we can have $s_1 = (r_{12}' \circ s_2) \oplus_1 C$. It is noteworthy that, C is also an aggregation function which only takes the negative part of a input set, and in order to share computation in this case C requires to be precomputed.

7.2.2 $f \in PU^{\odot}$. In this case, generally, f is not an injection because of the binary function $\odot$. We propose the splitting transformations for some cases, with which the previous solution can also be used.

First of all, the binary operators in PB are classified into two families, the addition family $ADD = \{+, -\}$ and the multiplication family $MUL = \{\times, /\}$. Without loss of generality, let $f = g_1(x) \odot g_2(x), g_1, g_2 \in PU^{\circ}$, then we can have following two splitting rules,

$$\sum g_1(x) \odot g_2(x) = \sum g_1(x) \odot \sum g_2(x), \odot \in ADD, \quad \text{(split1)}$$

$$\prod g_1(x) \odot g_2(x) = \prod g_1(x) \odot \sum g_2(x), \odot \in MUL. \quad \text{(split2)}$$

With these two rules, the original aggregation state $\sum_{\oplus} g_1(x) \odot g_2(x)$ is split into two aggregation states $\sum_{\oplus} g_1(x)$ and $\sum_{\oplus} g_2(x)$, for which the previous solution is still feasible. In another sense, the $\odot$ operator is pushed to the terminating function. For the left cases that are not covered by split rules, the current sharing condition is whether they have the identical expressions.

7.3 Equivalent expressions

Let $f(x) = f_1 \circ f_2^{-1}(x)$, and $f = p_l \circ ... \circ p_1, p_i \in PU, i \in (1, ..., l), l \in \mathbb{N}_{\geqslant 2}$. Note that every p_i is not the identity function and constant functions, otherwise it can be trivially removed, or f becomes a constant function. Such that, p_i can be one of the four different types of primitive functions, $(ax, log_a x, a^x, x^a)$. The type of the expression f is verified to check whether sharing is possible between s_1 and s_2. According to theorem 7.3, 7.4, 7.5, and 7.6, the expression f needs to be one of the following target types $(x, ax, a(log_b|x|), b^{ax}, x^a), a \in \mathbb{R}_{\neq 0}, b \in \mathbb{R}_{\neq 0,1}$, which is determined by $\oplus_1$ and $\oplus_2$ in s_1 and s_2. We propose a recursive method to answer this problem for the practical framework in this subsection.

In order to answer the question, all equivalent expressions of f need to be identified. However, generating all arbitrary equivalent expressions for a unary function may not be possible, and not all of them are useful for our question. From all the target types we can have that any of them is either a primitive unary function or a composition of two primitive unary functions, such that a target type has two properties: (1) it is an element in PU°; (2) its length is 1 or 2. Therefore, we concentrate on the equivalent expressions of f, which are also in PU° and have a length of 1 or 2. We formally define the equivalent expressions in the follows.

Definition 7.7. (**Equivalent expression**) Let f be a unary function in PU°, then the equivalent expression of f, $EE(f)$, is a set containing all unary functions f' which are also in PU°, satisfy $f'(x) = f(x)$, and have a length of 1 or 2,

$$EE(f) = \{f' | \forall x, f(x) = f'(x), f' \in PU^{\circ}, 1 \leqslant |f'| \leqslant 2\}.$$

We firstly illustrate the algorithm $G\_EE_2()$ for generating $EE(f)$, $|f| = 2$ by using the mathematical transformation of f, then we propose the recursive algorithm $G\_EE()$ for generating $EE(f), |f| \in \mathbb{N}_{>2}$.

Case 1: $G\_EE_2(f), |f| = 2$. Based on the target types, we exhaustively illustrate the mathematical transformations for the composition of a pair of the four types of primitive functions in table 4. Then, there only exist three different situations for the mathematical transformations: (1) merging transformations; (2) type changing; (3) no transforming. With this table, we define the function $Trans()$ generating the other equivalent expression for $f = p_2 \circ p_1$, and the function $T\_Trans()$ returning the type of the transformation. $Trans(p_2 \circ p_1)$ takes the type of p_2 and p_1, finds the corresponding one in table 4, and returns the other equivalent expression, while $T\_Trans(p_2 \circ p_1)$ returns the corresponding type of transformation. We take the first column of table 4 as an example for $Trans()$ and $T\_Trans()$ to illustrate the three different transforming situations,

- type merging: $Trans(ax \circ bx) = abx$ and $T\_Trans(ax \circ bx) = TM$;
- type changing: $Trans(ax \circ x^b) = x^b \circ a^{1/b}x$ and $T\_Trans(ax \circ x^b) = TC$;
- no transforming: $Trans(ax \circ b^x) = null$ and $T\_Trans(ax \circ b^x) = NT$.

With $Trans()$, we can have $G\_EE(f) = \{f, Trans(f)\}, |f| = 2$.

Case 2: $G\_EE(f), |f| \in \mathbb{N}_{>2}$. Let $|f| = l$, we propose the following recursive equation based on the associative property of functional composition.

$$EE(p_l \circ ... \circ p_1) = EE(f'_{l\to 2} \circ p_1), f'_{l\to 2} \in EE(p_l \circ ... \circ p_2). \quad (7)$$

Then we present the general steps of $G\_EE()$ in the follows. Firstly, for a unary function in PU°, we define two functions $getLast()$ getting the last primitive function and $rmLast()$ removing the last primitive function. For instance, if f is the input, then $getLast(f) = p_l$ and after $rmLast(f), f = p_{l-1} \circ ... \circ p_1$.

By recursively applying equation 7, the problem of generating $EE(f)$ in the general situation can be reduced to the case of generating $EE(p_l \circ p_{l-1})$, which can be generated by $G\_EE_2(p_l \circ p_{l-1})$. After having $EE(p_l \circ p_{l-1})$, every element $f'_{l\to l-1}$ in $EE(p_l \circ p_{l-1})$ is composed with p_{l-2} to generate $EE(f'_{l\to l-1} \circ p_{l-2})$, and all of the composition are merged together to have $EE(p_l \circ p_{l-1} \circ p_{l-2})$. Through repeating this procedure, $EE(f)$ will be eventually generated.

A key step in the backward phase of $G\_EE()$ is how to generate $EE(f'_{l\to l-1} \circ p_{l-2})$ with $f'_{l\to l-1}$ and p_{l-2}. We solve this problem by exhaustively analyzing the situations. It is trivial to see that if $|f'_{l\to l-1}| = 1$, then we can use $G\_EE_2(f'_{l\to l-1} \circ p_{l-2})$. For the other case $|f'_{l\to l-1}| = 2$, let $f'_{l\to l-1} = p'_l \circ p'_{l-1}$. Then a first transformation is called, that $Trans(p'_{l-1} \circ p_{l-2})$. As illustrated previously, there only exist three different types of transformations. For the case of type merging, $Trans(p'_l, Trans(p'_{l-1} \circ p_{l-2}))$ can be directly called. For the case of no transforming, we can have $p'_l \circ p'_{l-1} \circ p_{l-2}$ is the only transformation of $p'_l \circ p'_{l-1} \circ p_{l-2}$, and the length is bigger than 2, such that this candidate is deprecated. For the case of type changing, we continue the second transformation, that $Trans(p'_l \circ getLast(Trans(p'_{l-1} \circ p_{l-2}))$, of which based on the different cases of the outputs the above procedure can be repeated. Assuming that both the first and second transformation are type changing and we have an output $p''_l \circ p''_{l-1} \circ p'_{l-2}$, then another transformation

Table 4: Mathematical transformations of $p \circ p'(x), p(x), p'(x) \in PU_{\neq identity, constant}$.

$p'(x)$ \ $p(x)$	$ax, a \neq 1, 0$	$x^a, a \neq 0$	$log_a\|x\|, a > 0, a \neq 1$	$a^x, a > 0, a \neq 1$
$bx, b \neq 1, 0$	ax ∘ bx = abx, or x, ab = 1	$\mathbf{x^a \circ bx = b^a x \circ x^a}$	$log_a\|x\| \circ bx = log_a\|x\| \circ bx$	$a^x \circ bx = (a^b)^x$
$x^b, b \neq 0$	$\mathbf{ax \circ x^b = x^b \circ a^{1/b}x, a > 0}$	$x^a \circ x^b = x^{ab}$, or x, ab = 1	$\mathbf{log_a\|x\| \circ x^b = bx \circ log_a\|x\|}$	$a^x \circ x^b = a^x \circ x^b$
$log_b x, b > 0, \neq 1$	$\mathbf{ax \circ log_b\|x\| = log_b\|x\| \circ x^a}$	$x^a \circ log_b\|x\| = x^a \circ log_b\|x\|$	$log_a\|x\| \circ log_b\|x\| = log_a\|x\| \circ log_b\|x\|$	$a^x \circ log_b x = a^{log_b x}$,or |x|, a = b
$b^x, b > 0, \neq 1$	$ax \circ b^x = ax \circ b^x$	$x^a \circ b^x = (b^a)^x$	$log_a\|x\| \circ b^x = (log_a b)x$, or x, log_ab = 1	$a^x \circ b^x = a^x \circ b^x$

is called $Trans(p''_{l-1} \circ p'_{l-2})$. If this transformation is not the type merging case, then the length can not be reduced to 2. Such that, the candidate $p'_l \circ p'_{l-1} \circ p_{l-2}$ is also deprecated. Finally, if for every $f'_{l \to l-1} \in EE(p_l \circ p_{l-1})$, $EE(f'_{l \to l-1} \circ p_{l-2})$ is still empty, we have $EE(p_l \circ ... \circ p_1)$ is empty. Otherwise, the following primitive function p_{l-3} will be taken.

7.4 Complexity of sharing

In order to verify the sharing possibilities between a given aggregation state s_1 and a cached aggregation state s_2, both of them will be given as the input for $G\_EE()$. If there are more than one aggregation state cached in memory, they will be verified in turn until one is satisfied. Therefore the sharing complexity is related to two factors: (1) the complexity of $G\_EE()$, (2) the number of cached aggregation states,

We firstly analyze the time complexity of $G\_EE()$. For $f = p_l \circ .... \circ p_1$, in worst cases, the recursive algorithm $G\_EE(f)$ proceed one time of $Trans()$ for one continuous pair of primitive function, which can be $l - 1$ in total. Such that the complexity of $G\_EE(f)$ is $O(l)$. Secondly, it is trivial to see that if m aggregation states are cached in memory, then the complexity for the second step is $O(m)$. Therefore, in the case that the length of the unary function f_1 in s_1 is l_1, and m different aggregation states s_2 cached in memory, where every one of them contains a unary function with the length $l_2^i, i \in (1, ..., m)$, then the total sharing complexity can be $\Omega(l_1 + l_2^i)$ in best cases and $O(ml_1 + \sum_{i=1}^{m} l_2^i)$ in worst cases.

7.5 Symbolic index

In this subsection, we propose the symbolic index to organize the cached aggregation states, such that the total sharing complexity can be decreased to $\Omega(1)$ in best cases and $O(l_1 + logl_0 + m)$ in worst cases, where l_0 is the maximum length of aggregation state cached in memory.

Observation According to theorem 7.3,7.4,7.5 and 7.6, we mainly need to check whether $f_1 \circ f_2^{-1}(x)$ or $f_1 \circ f_2^{-1}(u), u = sign(x)$, is a specific type of unary functions, and algorithm 2 only needs the type of primitive functions in $f_1 \circ f_2^{-1}(x)$ or $f_1 \circ f_2^{-1}(u)$. Therefore, if the type data of unary functions is stored, this type-checking phase can be preprocessed. Moreover, the computation results of aggregation state s_2 can only be reused for the elements in its derivable set $D(s_2)$, and derivable sets can be equivalent because the reusing functions r_{12} in theorem 7.3 and 7.5 are injections. Therefore, the searching space m can be decreased by organizing the cached aggregation states using derivable set and equivalent derivable sets.

We firstly propose the notion of symbolic primitive functions to store the type data of primitive functions. Computing every primitive function besides constant functions requires inputs of an element of base data x and a parameter a. x is given at the time of scanning base data, while a is received at the time of declaring functions. For instance, linear functions have the shape of ax, which takes x and a and proceeds a simple multiplication on them. We define symbolic primitive function in the follows.

Definition 7.8. (**Symbolic primitive function**) A symbolic primitive function is a function $p^{(a)}(x)$ taking two real numbers a and x as the input and return a real number as the output, and for a constant a', $p^{(a')}(x) \in PU$.

Then the symbolic aggregation state is formally defined in the follows.

Definition 7.9. (**Symbolic aggregation state**) A symbolic aggregation state $sy\_s(X, \bar{a})$ is an aggregation function taking two inputs, a set of real numbers X and a sequence of real numbers $\bar{a} = (a_l, ..., a_1)$, $sy\_s(X, \bar{a}) = \sum_{\oplus, x \in X} p_l^{(a_l)} \circ ... \circ p_1^{(a_1)}(x), \oplus \in (+, \times), l \in \mathbb{N}_{>1}$, where $p_i^{(a_i)}(x), i \in (1, ..., l)$ is a symbolic primitive function.

A symbolic aggregation state sy_s can be generated from a concrete one s. A concrete aggregation state can be also obtained from a symbolic aggregation sate and a sequence $\bar{a} = (a_l, ..., a_1)$. Such that, based on the definition of concrete derivable set, we can have the symbolic derivable set in the follows,

$$D(sy\_s) = \{sy\_s' | \exists \bar{a}', \bar{a}, \forall X, sy\_s'(X, \bar{a}') = r \circ sy\_s(X, \bar{a})\}.$$

For a fixed length l_0, the following symbolic derivable set of a symbolic aggregation state $sy\_s, |sy\_s| \leqslant l_0$, can be built,

$$D_{l_0}(sy\_s) = \{sy\_s' | sy\_s' \in D(sy\_s), |sy\_s'| \leqslant l_0\}.$$

We explain the reason in the follows. Although there exists infinitive concrete aggregation states $s, |s| \leqslant l_0$, the symbolic ones $sy\_s, |sy\_s| \leqslant l_0$ is countable, of which the total number in the practical sharing framework is $2 \times (4^{l_0} + ... + 4^1) + 2$, because two types of $\oplus$ functions, four types of primitive functions, and two additional ones, $\sum x$ and $\prod x$. Moreover, symbolic aggregation states indeed contain the type information of $\oplus$ and unary functions. Such that, the sharing possibility of sy_s and sy_s' can be verified by $G\_EE()$. Therefore, for a given $sy\_s, |sy\_s| \leqslant l_0$, we take any $sy\_s', |sy\_s'| \leqslant l_0$, and sy_s as the input of $G\_EE()$ to generate $D_{l_0}(sy\_s)$. With this intuition, we propose the following symbolic index.

Index structure. A symbolic index is a set of tables $\{T_{l_0}, ..., T_1\}$. Every $T_i, i \in (1, ..., l_0)$, has five columns (ID, Symbolic Aggregation State, Symbolic Derivable Set, Sharing Candidate, Concrete Aggregation State). In the column symbolic aggregation state, every row of T_i stores the information of one symbolic aggregation state $sy\_s, |sy\_s| = i$. The column ID stores the encoded number of sy_s.

Table 5: T_1 in the symbolic index.

ID	sy_s	$D_{l_0}(sy\_s)$	$SC(sy\_s)$	s
-2	$\sum x$	$\{\sum x, \sum a_1 x, \prod a_1^x, ...\}$	$\{\sum x\}$	$\{R\}$
-1	$\prod x$	$\{\prod x, \prod x^{a_1}, \sum log_{a_1}\|x\|, ...\}$	$\{\prod x, ...\}$	$\{R\}$
0	$\sum a_1 x$	$\{\sum a_1 x, \sum x, \prod a_1^x, ...\}$	$\{\sum x\}$	null
1	$\sum log_{a_1}\|x\|$	$\{\sum log_{a_1}\|x\|, ...\}$	$\{\prod x, ...\}$	$\{(\bar{a}, R), ...\}$
2	$\sum a_1^x$	$\{\sum a_1^x, ...\}$	$\{\sum a_1^x, ...\}$	$\{(\bar{a}, R), ...\}$
3	$\sum x^{a_1}$	$\{\sum x^{a_1}, ...\}$	$\{\sum x^{a_1}, ...\}$	$\{(\bar{a}, R), ...\}$
4	$\prod a_1 x$	$\{\prod a_1 x, ...\}$	$\{\prod a_1 x, ...\}$	$\{(\bar{a}, R), ...\}$
5	$\prod log_{a_1}\|x\|$	$\{\prod log_{a_1}\|x\|, ...\}$	$\{\prod log_{a_1} x, ...\}$	$\{(\bar{a}, R), ...\}$
6	$\prod a_1^x$	$\{\prod a_1^x, \sum x, \sum a_1 x, ...\}$	$\{\sum x\}$	null
7	$\prod x^{a_1}$	$\{\prod x^{a_1}, ...\}$	$\{\prod x, ...\}$	$\{(\bar{a}, R), ...\}$

The column symbolic derivable set stores the $D_{l_0}(sy\_s)$ of sy_s. The column sharing candidate simply store all sharing candidates for sy_s, which is a set $SC(sy\_s) = \{sy\_s' | sy\_s \in D_{l_0}(sy\_s')\}$. The last column is a set of pairs $\{(\bar{a}, R), ...\}$ for concrete aggregation states, where R is the computation result of a concrete aggregation state, and the concrete aggregation state is generated by the symbolic state and the sequence $\bar{a}$. An example T_1 is illustrated in table 5, where only subsets are contained in the column symbolic derivable set and sharing candidate.

Index building. Building a symbolic index contains for every row in every table $T_i, i \in (1, ..., l_0)$, generating the four attributes, symbolic aggregation states, ID, symbolic derivable sets and sharing candidates, and sorting rows according to ID in T_i. Let $sy\_s(X, \bar{a}) = \sum_{\oplus, x \in X} p_i^{(a_i)} \circ ... \circ p_1^{(a_1)}(x), i \in (1, ..., l_0)$. Firstly, symbolic aggregation states are obtained by selecting one type of $\oplus$ from $(+, \times)$ and one type of $p_j^{(a_j)}, j \in (1, ..., i)$ from linear, logarithm, exponential, power, which will produce 2×4^i rows in T_i. And in the case of $i = 1$, two addition ones are added, $\sum x$ and $\prod x$. Moreover, all the rows in T_i are classified into two parts with $\oplus = +$ and $\oplus = \times$, and we define offset$(+) = 0$ and offset$(\times) = 4^i$. Then, for every type of linear, logarithm, exponential, power, a encoded integer is assigned, that $en(linear) = 0$, ... and $en(power) = 3$. Then the encoded function $EN()$ is defined to generate the ID,

$$EN(sy\_s) = en(p_i^{(a_i)}) \times 4^{i-1} + ... + en(p_1^{a_1}) \times 4^0 + offset(\oplus). \quad (8)$$

For the special two cases $\sum x$ and $\prod x$, we assign their ID as -2 and -1. Thirdly, sy_s is added in $D_{l_0}(sy\_s)$, and the function $G\_EE()$ is ran with the input of any combination of sy_s and another symbolic aggregation state from $(T_1, ..., T_{l_0})$ to generate $D_{l_0}(sy\_s)$. At fourth, we iterate on the column of symbolic derivable set to find identical ones, where only the symbolic state with the shortest length will be put in the sharing candidate column. Otherwise, symbolic aggregation state sy_s will be registered in the sharing candidate column of every element in $D_{l_0}(sy\_s)$. Finally, for every table T_i, all the rows are sorted based on the ID column.

It is noteworthy that, the sharing candidates column may store one or several symbolic aggregation states. In the latter case, all the sharing candidates are sorted based on two criteria: (1) whether a sharing candidate contains a concrete computation; (2) the length of the sharing candidates. And the first criteria always has the higher priority than the second one. This machinery can help find a sharing candidate that can be feasible to reuse in $O(1)$ time.

Index searching. Given an aggregation state s, firstly the following three types of data are retrieved from s, the length of $|s|$, symbolic aggregation state sy_s, and the sequence $\bar{a}$. Then according to $|s|$, the corresponding table $T_i, i = |s|$ is located by a first binary search. Furthermore, based on the encoded formula (equation 8), the ID of sy_s is computed. Following this, a second binary search is proceeded to find sy_s in T_i. Afterward, if the first sharing candidate has a concrete computing result, it will be retrieved for computing s, otherwise, the real computation will be launched. Which concrete computation is launched will depend on the number of sharing candidates for sy_s. If sy_s contains only one sharing candidate, then the only sharing candidate will be launched to compute, e.g. the row with ID=0 and ID=6 in table 5. Otherwise, the current sy_s with the received $\bar{a}$ will be computed. Finally, the result of s is returned.

Index updating. If a real computation of a concrete aggregation state s is launched, an index update is proceeded because of the sorting machinery in the sharing candidate column. Updating a symbolic index is simply telling every element in $D_{l_0}(sy\_s)$ that sy_s has a concrete computation, such that sy_s will be ranked nearer to the front in the set of sharing candidates of every element in $D_{l_0}(sy\_s)$.

Search complexity. For an aggregation state $s, |s| = i, i \leqslant l_0$, the searching time complexity of sy_s in the symbolic index can be $\Omega(1)$ in best cases and $O(i + logl_0 + m)$ in worst cases (m concrete aggregation stated results cached in memory). We explain the analysis in the follows. The first search is locating the table T_i, which can take $\Omega(1)$ in best cases and $O(logl_0)$ in the worst cases. The second search is finding sy_s in T_i, which can take $\Omega(1)$ in best cases and $O(log|T_i|)$ in worst cases, where $|T_i|$ is the total number of rows in T_i, $|T_i| = 2 \times 4^i$, then $O(log|T_i|) = O(i)$. If a sharing candidate contains only one concrete computation result, it can be directly retrieved. While, if all the m concrete results are cached in the sharing candidate, it will take $O(m)$ to locate the required one. Therefore, the best cases can take $\Omega(1)$, while the worst cases can take $O(i + logl_0 + m)$.

Building and space complexity. The building time complexity of the symbolic index is $O(4^{3l_o})$. The total number of rows in all tables $T_i, i \in (l_0, ..., 1)$, are $\frac{4^{l_0+1} - 2}{3}$. Note that, if the practical framework is not extended, the symbolic index will remain unchanged, then it is only necessary to build it one time. Such that, once the symbolic index is built, it can be serialized to disk and be deserialized to memory at the running time.

7.6 Pull-up rules

Choosing a bigger length l_0 of the symbolic index may still consume some memory, and the tables with bigger lengths may not be often used based on the composition of realistic functions. Therefore, we propose a hybrid solution, that we fix a reasonable length l_0 for generating the symbolic index, and for aggregation states with length $l \leqslant l_0$, the symbolic index is used to share computation. For the left cases $l > l_0$, instead of directly verifying sharing condition with cached elements, we propose pull-up rules to reduce the length l to l', then if $l' \leqslant l_0$, aggregation states will be passed to symbolic index, otherwise the sharing conditions are verified. The purpose

of this method can be summarized that reducing aggregation state length and using the symbolic index in an exhaustive way.

The intuition of reducing the length of aggregation states is pushing some unary functions to terminating function in the formal framework. Specifically, for an aggregation state $s = \sum_{\oplus} p_l \circ ... \circ p_1, p_i \in PU, i \in (1, ..., l)$, if s can be transformed to $p_l \circ \sum_{\oplus'} p_{l-1} \circ .. \circ p_1$, then p_l can be pulled up to terminating functions and the length of s is reduced to $l-1$. This process can be repeated for p_{l-1} and the follows if possible, until a minimum length is obtained.

We propose the complete pull-up conditions. From the exhaustive combination of $\oplus$ and $\oplus'$ in the practical scope, that $(\sum, \sum)$, $(\sum, \prod)$, $(\prod, \prod)$, and $(\prod, \sum)$, we propose the following four pull-up rules for each of them to pull one primitive function up.

$$\sum ax \ \rightarrow \ ax \circ \sum x, \ a \in \mathbb{R}_{\neq 1,0}. \tag{R2}$$

$$\sum log_a x \ \rightarrow \ \log_a x \circ \prod x, a \in \mathbb{R}_{>0,\neq 1}. \tag{R3}$$

$$\prod x^a \ \rightarrow \ x^a \circ \prod x, a \in \mathbb{R}_{\neq 0}. \tag{R4}$$

$$\prod a^x \ \rightarrow \ a^x \circ \sum x, a \in \mathbb{R}_{\neq 0,1}. \tag{R5}$$

From theorem 7.3,7.4,7.6 and 7.5, we can have that these four rules are the complete conditions for pulling one primitive function up. For instance, $R2$ describe the transformation $\sum ax = ax \circ \sum x$, which can also be considered as the case of sharing computation between $\sum ax$ and $\sum x$, and from theorem 7.3 we know only ax is possible.

Counting for free If the computation of the aggregation function count is admitted for free, we can have an additional transformation, $\prod ax = \prod a \times \prod x = a^{count} x \circ \prod x$. Generally, $a^{count} x$ is a bivariate function instead of a unary one (this is the reason why it is not identified in theorem 7.6). However, with the stored count value, the function a^{count} can be computed and $a^{count} x$ is changed to a linear function because a^{count} is a constant at this moment. Moreover, due to the fact *count* is necessary for frequently used aggregate functions e.g *average*, and priori data statistics for query optimization, the aggregation count is computed for free. Then, we propose the following transformation rule $R6$, which is also a kind of pull-up rules due to its feature of pulling a 'linear function' up.

$$\prod ax = (a^{count} x) \circ \prod x, a \in \mathbb{R}_{\neq 1,0}. \tag{R6}$$

Supplementary pull-up rules. We propose supplementary pull-up rules to transform a non-feasible pull-up candidate to a feasible one. In fact, in the cases that s contains several primitive functions and the last primitive function p_l is not a feasible pull-up candidate, the length of s may be still possible decreased because of the type changing transformation of primitive functions. We exhaustively analyze these supplementary pull-up rules in the following five exhaustive cases.

Case 1: $\oplus = +$, *and exploring supplementary transforming candidates for* $R2$. When $\oplus = +$, then if p_l is linear or logarithm, then $R2$ and $R3$ can be applied. Therefore, the non-feasible cases are power and exponential. And we are searching candidates for $R2$, then the target type is linear.

When p_l is a power function, we firstly identify that power function and linear function has the reordering feature (see in table 4), that $x^a \circ bx = b^a x \circ x^a$. Therefore, we propose the transforming rule $R6$, which we call the *linear-up* rule due to its linear-type reordering feature,

$$power \circ linear \leftrightarrow linear \circ power. \tag{R7}$$

By applying $R7$, a linear function can still be pulled up by $R2$. Moreover, power functions can absorb power functions, that $x^a \circ x^b = x^{ab}$. Then several continuous power functions and a linear function is also feasible for $R7$. According to the second column in table 4, $R7$ is the only type changing when p_l is a power function. Therefore, no matter how to transform the following functions, if there does not exist linear functions behind power functions, then the transformations are still not interesting for $R2$. We also found two cases that can produce linear functions in table 4, which can be the supplementary rules for $R7$. Transforming rules $R8$ and $R9$ are proposed to cover these two sub-situations, which we call the *linear-birth* rules due to its feature of two non-linear functions producing a linear function.

$$logarithm \circ exponential \rightarrow linear, \tag{R8}$$

$$logarithm \circ power \rightarrow linear \circ logarithm. \tag{R9}$$

Such that, sequential applying $R8, R7, R2$ or $R9, R7, R2$, can still pull up some functions. And $R7, R8, R9$ are the complete supplementary rules for $R2$.

When p_l is an exponential function, according to the fourth column in table 4, no matter how to transform primitive functions, the type of the first function cannot be changed. Therefore, none functions can be pulled up.

Case 2: $\oplus = +$, *and exploring supplementary transforming candidates for* $R3$. Similarly, we are searching candidates for $R3$, then p_l can be a power or exponential function and the target type is logarithm. However, according to the second and fourth column in table 4, no matter p_l is an exponential or power function, none transformations can change the type of p_l to be $logarithm$. Therefore, in this case, it is impossible to have transforming candidates for $R3$.

Similarly, in the *case 3:* $\oplus = \times$, *exploring supplementary rules for* $R4$, and the *case 4:* $\oplus = \times$, *exploring supplementary rules for* $R5$, there does not exist supplementary transforming candidates.

Case 5: $\oplus = \times$, *and exploring supplementary transforming candidates for* $R6$. In this case, we are searching candidates for $R6$, and p_l is a logarithmic function and the target type is linear. Two complete cases can be identified based on linear-birth rules $R8$ and $R9$ (see the third column of table 4). Moreover, because power functions can absorb power functions and can be reordered with linear functions, therefore these two patterns can be also supplementary transformations of $R9$ in this case, $log_a x \circ x^{b_1} \circ x^{b_2} = log_a x \circ x^{b_1 b_2} = b_1 b_2 x \circ log_a x$, and $log_a x \circ b_1 x \circ x^{b_2} = log_a x \circ x^{b_2} \circ b_1^{1/b_2} x = b_2 x \circ log_a x \circ b_1^{1/b_2} x, b_1 > 0$. Therefore, in this case, by sequential applying $R8, R6$, or $R9, R6$, or $R7, R8, R6$, the length of aggregation state can still be decreased.

The application of pull-up rules has linear time complexity $O(l)$, because it visits each primitive function one time.

8 CONCLUSION AND ONGOING WORK

We analyze how to efficiently decompose user-defined aggregation functions and exhaustively reuse their computation results.

A generic framework for aggregation functions is firstly identified, which is mapped to the $\mathcal{MRC}$ model to generate the efficient and generic algorithm $MR(\alpha)$. Then, we analyze the effects of two widely used $\oplus$ functions in practice on $MR(\alpha)$, + and $\times$.

On the side of sharing aggregation computation, we concentrate on reusing partial aggregation results, which is more generic than total aggregation result. Then we illustrate the sharing problem is undecidable for arbitrary partial aggregation. Hereafter, we identify several primitive classes of unary and binary functions that are widely used in practical aggregation functions. Then, for this practical framework, the complete sharing condition is proposed. Consequently, we propose the symbolic index and pull-up rules to reduce the sharing complexity.

Based on the practical framework of UDA and corresponding theoretical propositions, we are implementing the library DS4Alpha aiming at providing a systematic approach to process user-defined aggregation in distributed computing. Specifically, DS4Alpha provides a 'declarative' interface for defining aggregation like writing its mathematical formula. Then, the execution plan of every received aggregation will be transformed and checked to verify the sharing possibilities with the cached aggregation results in the system. In the case of none results can be reused, the execution plan will be materialized to functions, which are sent to slave nodes to launch the computation. Until now, according to the outline (see figure 1), the declarative interface and UDA parser have already been implemented, and we will be devoted into the UDA optimizer. Finally, we will compare DS4Alpha to user-defined aggregation API in main-stream cluster computing framework, e.g. Apache Spark UDAF and Aggregator, in terms of productiveness and performance.

REFERENCES

[1] [n. d.]. DB2 built-in functions. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/sqlref/src/tpc/db2z_sqlfunctionsintro.html. ([n. d.]).

[2] [n. d.]. Miscrosoft SQL functions. https://docs.microsoft.com/en-us/sql/t-sql/functions/functions. ([n. d.]).

[3] [n. d.]. PostgreSQL. https://www.postgresql.org/docs/current/static/functions-aggregate.html. ([n. d.]).

[4] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *Pod. '13*, Vol. 64. 273.

[5] C.Liu, J.Zhang, H.Zhou, S.McDirmid, Z.Guo, and T.Moscibroda. 2014. Automating distributed partial aggregation. In *SOCC'14*. 1–12.

[6] Sara Cohen. 2006. User-defined aggregate functions: bridging theory and practice. In *SIGMOD'06*. 49–60.

[7] SARA COHEN, W.NUTT, and Y.SAGIV. 2006. Rewriting queries with arbitrary aggregation functions using views. *ACM TODS* 31, 2 (June 2006), 672–715.

[8] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A stream database for network applications. In *Proc. 2003 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '03*. ACM Press, New York, New York, USA, 647. https://doi.org/10.1145/872757.872838

[9] Algredo Cuzzocrea. 2015. Aggregation and multidimensional analysis of big data for large-scale scientific applications: models, issues, analytics, and beyond. In *SSDBM'15*.

[10] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, Jeffrey F. Naughton, Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching multidimensional queries using chunks. In *Proc. 1998 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '98*, Vol. 27. ACM Press, New York, New York, USA, 259–270. https://doi.org/10.1145/276304.276328

[11] Jon Feldman, S.muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. 2010. On Distributing Symmetric Streaming Computations. *ACM TALG* 6, 4 (August 2010).

[12] M.T. Goodrich, N. Sitchinava, and Q. Zhang. 2011. Sorting, Searching, and Simulation in the MapReduce Framework. In *ISAAC'11 Proceedings of the 22nd international conference on Algorithms and Computation*. 374–383.

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

[14] Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. 2011. Optimized processing of multiple aggregate continuous queries. In *Proc. 20th ACM Int. Conf. Inf. Knowl. Manag. - CIKM '11*. ACM Press, New York, New York, USA, 1515. https://doi.org/10.1145/2063576.2063793

[15] Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing data cubes efficiently. In *Proc. 1996 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '96*, Vol. 25. ACM Press, New York, New York, USA, 205–216. https://doi.org/10.1145/233269.233333

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

[17] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. 2007. Sharing Aggregate Computation for Distributed Queries. In *Proc. 2007 ACM SIGMOD Int. Conf. Manag. Data (SIGMOD '07)*. ACM, New York, NY, USA, 485–496. https://doi.org/10.1145/1247480.1247535

[18] J.Gray, A.Bosworth, A.Layman, and H.Pirahesh. 1997. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery* 1, 1 (January 1997), 29–53.

[19] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA'10*. 938–948.

[20] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. *Proc. 2006 ACM SIGMOD Int. Conf. Manag. data* (2006), 623–634. https://doi.org/10.1145/1142473.1142543

[21] M.Jean-Luc and T.Bruno. 2015. Preassociative Aggregation Functions. *Fuzzy Sets and Systems* 268 (June 2015), 15–26.

[22] M.Jean-Luc and T.Bruno. 2016. Strongly Barycentrically Associative and Preassociative Functions. *Fuzzy Sets and Systems* 437, 1 (May 2016), 181–193.

[23] S.Madden, M.J.Franklin, J.M.Hellerstein, and W.Hong. 2002. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI'02*. 131–146.

[24] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record* 26, 1 (March 1997), 65–74.

[25] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data Canopy: Accelerating Exploratory Statistical Analysis. In *ACM SIGMOD International Conference on Management of Data*.

[26] Y.Yu, M.Isard, and P.Kumar Gunda. 2009. Distributed Aggregation For Data-Parallel Computing: Interfaces and Implementations. In *SOSP'09*. 247–260.

[27] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. 2005. Multiple aggregations over data streams. In *Proc. 2005 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '05*. ACM Press, New York, New York, USA, 299. https://doi.org/10.1145/1066157.1066192

Table 6: Symmetric aggregation functions in canonical forms.

Aggregation	Formula	$F(x_i)$	$F(x_i) \oplus F(x_j)$	$T(S)$
Max	–	id	max	id
Min	–	id	min	id
Sum	$\sum x_i$	id	$+$	id
Count	$\sum 1$	id	$+$	id
Avg μ	$\frac{\sum x_i}{n}$	$(id, 1)$	$(+,+)$	$\frac{s_1}{s_2}$
Variance σ^2	$\frac{\sum(x_i-\mu(X))^2}{n}$	$((x_i-\mu(X))^2, 1)$	$(+,+)$	$\frac{s_1}{s_2}$
Std_variance σ	$\sqrt{\frac{\sum(x_i-\mu(X))^2}{n}}$	$((x_i-\mu(X))^2, 1)$	$(+,+)$	$\sqrt{\frac{s_1}{s_2}}$
Covariance $cov(X,Y)$	$\frac{\sum(x_i-\mu(X))(y_i-\mu(Y))}{n}$	$((x_i-\mu(X))(y_i-\mu(Y)), 1)$	$(+,+)$	$\frac{s_1}{s_2}$
Correlation $corr(X,Y)$	$\frac{cov(X,Y)}{(\sigma(X)\sigma(Y))}$	$((x_i-\mu(X))^2, (y_i-\mu(Y))^2, (x_i-\mu(X))(y_i-\mu(Y)), 1)$	$(+,+,+,+)$	$\frac{s_3/s_4}{\sqrt{s_1/s_4}\sqrt{s_2/s_4}}$
$corr(X,Y)^2$	$\frac{cov(X,Y)}{(\sigma(X)\sigma(Y))}$	$((x_i-\mu(X))^2, (y_i-\mu(Y))^2, (x_i-\mu(X))(y_i-\mu(Y)), 1)$	$(+,+,+,+)$	$(\frac{s_3/s_4}{\sqrt{s_1/s_4}\sqrt{s_2/s_4}})^2$
Sum of squares	$\frac{\sum x_i^2-(\sum x_i)^2}{n}$	$(x_i^2, id, 1)$	$(+,+,+)$	$\frac{s_1-s_2}{s_3}$
Sum of squares(X,Y)	$\frac{\sum x_i \times y_i-(\sum x_i)\times(\sum y_i)}{n}$	$(x_i \times y_i, x_i, y_i, 1)$	$(+,+,+)$	$\frac{s_1-s_2\times s_3}{s_4}$
Cumulative distance(r)	$\frac{count_{x_i \leqslant r}}{count}$	$(x_i \leqslant r ? 1:0, 1)$	$(+,+)$	$\frac{s_1}{s_2}$
First_value	–	id	First_value()	id
Last_value	–	id	Last_value()	id
Median	–	id	$\cup$	$median$
Percentile	–	id	$\cup$	$percentile$
Rank	–	id	$\cup$	$rank$
Product	$\prod x_i$	id	$\times$	id
Geometric_mean	$(\prod x_i)^{1/n}$	$(id, 1)$	$(\times,+)$	$(s_1)^{1/s_2}$
Power_mean	$(\frac{\sum(x_i)^p}{n})^p$	$(x_i^p, 1)$	$(+,+)$	$(\frac{s_1}{s_2})^p$
Skewness	$\frac{(\sum(x-\mu)^3)/n}{((\sum(x-\mu)^2)/n)^{3/2}}$	$((x_i-\mu(X))^3, (x_i-\mu(X))^2, 1)$	$(+,+,+)$	$\frac{s_1/s_3}{(s_2/s_3)^{3/2}}$
Kurtosis	$\frac{(\sum(x-\mu)^4)/n}{((\sum(x-\mu)^2)/n)^2}$	$((x_i-\mu(X))^4, (x_i-\mu(X))^2, 1)$	$(+,+,+)$	$\frac{s_1/s_3}{(s_2/s_3)^2}$
LogSumExp	$ln(\sum exp(x_i))$	$exp(x_i)$	$+$	ln
TF(term:t,doc:d)	$\frac{count_t}{count}$	$(t ? 1:0, 1)$	$(+,+)$	$\frac{s_1}{s_2}$
IDF(term:t,corpus:D)	$Log(\frac{count}{count_t d})$	$(1, t \in d_j ? 1:0)$	$(+,+)$	$Log(\frac{s_1}{s_2})$
TF-IDF(t,d,D)	TF $\times$ IDF	$(d:(t ? 1:0, 1), D:(1, t \in d_j ? 1:0, 1))$	$(+,+,+,+)$	$\frac{s_1}{s_2} \times Log(\frac{s_3}{s_4})$