Efficient Data-Parallel Cumulative Aggregates for Large-Scale Machine Learning

Matthias Boehm[1] Alexandre V. Evfimievski[2] Berthold Reinwald[3]

Abstract: Cumulative aggregates are often overlooked yet important operations in large-scale machine learning (ML) systems. Examples are prefix sums and more complex aggregates, but also preprocessing techniques such as the removal of empty rows or columns. These operations are challenging to parallelize over distributed, blocked matrices—as commonly used in ML systems—due to recursive data dependencies. However, computing prefix sums is a classic example of a presumably sequential operation that can be efficiently parallelized via aggregation trees. In this paper, we describe an efficient framework for data-parallel cumulative aggregates over distributed, blocked matrices. The basic idea is a self-similar operator composed of a forward cascade that reduces the data size by orders of magnitude per iteration until the data fits in local memory, a local cumulative aggregate over the partial aggregates, and a backward cascade to produce the final result. We also generalize this framework for complex cumulative aggregates of sum-product expressions, and characterize the class of supported operations. Finally, we describe the end-to-end compiler and runtime integration into SystemML, and the use of cumulative aggregates in other operations. Our experiments show that this framework achieves both high performance for moderate data sizes and good scalability.

Keywords: Cumulative Aggregates, ML Systems, Large-Scale Machine Learning, Data-Parallel Computation, Apache SystemML

1 Introduction

Large-Scale ML Systems: Machine learning (ML) and artificial intelligence in general are transforming our lives from recommendations and driving assistants to finance, health care, and enterprise ML. Large data collections are essential for learning these ML models [Co09; Da10], especially for complex models with many parameters. Thanks to feedback loops for label acquisition like active learning [CAL94; CGJ94; RMJ06] and weak supervision techniques [Ra16; Tr18], there is also abundant labeled data. These large labeled data collections in turn necessitate large-scale ML systems with the ability for distributed computation whenever necessary. State-of-the-art large-scale ML systems, therefore, follow either of two predominant distributed architectures: data-parallel operations for batch algorithms, often on top of data-parallel computation frameworks such as MapReduce [DG04], Spark [Za12], or Flink [Al14]; or parameter servers for mini-batch algorithms

[1] Graz University of Technology, Graz, Austria, m.boehm@tugraz.at
[2] IBM Research – Almaden, San Jose, USA, evfimi@us.ibm.com
[3] IBM Research – Almaden, San Jose, USA, reinwald@us.ibm.com

[De12]. In SystemML [Bo14; Bo16], we support the general case of local and distributed data-parallel operations, task-parallel parfor loops, as well as local and distributed parameter servers, which can be seamlessly combined in a high-level language with R-like syntax.

Cumulative Aggregates: While data-parallel matrix multiplication [Bo16; Gh11; HBY13; Ro17; YSC15; Yu17], element-wise operations and aggregations [Bo18], meta learning [Sc15], and statistical functions [TTR12] have received lots of attention, important and challenging distributed operations like cumulative aggregates are largely overlooked in the literature. An example is the computation of column-wise prefix sums via

$$\mathbf{Z} = \text{cumsum}(\mathbf{X}) \quad \text{with} \quad \mathbf{Z}_{ij} = \Sigma_{k=1}^{i} \mathbf{X}_{kj} = \mathbf{X}_{ij} + \mathbf{Z}_{(i-1)j}, \tag{1}$$

where $\mathbf{Z}$ and $\mathbf{X}$ are $n \times m$ matrices, and output cells $\mathbf{Z}_{ij}$ are defined as the column-wise sum from 1 through i, or recursively (with $\mathbf{Z}_{0j} = 0$). Other common cumulative aggregates include cummin($\mathbf{X}$), cummax($\mathbf{X}$), cumprod($\mathbf{X}$) for cumulative minima, maxima, and products. Applications of cumulative aggregates include (1) iterative algorithms for survival analysis such as Cox hazard regression or Kaplan-Meier, (2) spatial and structural data processing via linear algebra, and (3) data preprocessing such as the sub-sampling of rows or the removal of empty rows, which both internally rely on cumulative aggregates.

Parallel Cumulative Aggregates: Although the recursive formulation in Equation (1) seems inherently sequential, computing prefix sums is a classic example of an operation that allows for efficient parallelization via aggregation trees [Bl93; CBZ90]. The basic idea is a logical tree over the input data, level-wise parallel aggregation (up sweep) and redistribution of offsets (down sweep) to compute the final output. This general approach of parallel cumulative aggregates is broadly applicable, but the integration into large-scale ML systems—with unordered distributed datasets—has received little attention so far.

Contributions: Our primary contribution is a holistic description of a framework for distributed, data-parallel cumulative aggregates and its integration in Apache SystemML, which is representative for state-of-the-art, large-scale ML systems:

- *Background:* In Section 2, we provide background on SystemML's compiler and matrix representations, and introduce straw-man solutions for cumulative aggregates.
- *DistCumAgg Framework:* In Section 3, we then describe our self-similar framework for distributed cumulative aggregates. Besides basic aggregates, we also introduce complex cumulative aggregates for common sum-product recurrence expressions.
- *System Integration:* Furthermore, in Section 4, we describe the end-to-end compiler and runtime integration of the DistCumAgg framework in SystemML, and the use of cumulative aggregates in other operations such as the removal of empty rows.
- *Experiments:* Finally, in Section 5, we report on experiments in SystemML, including baseline comparisons with R, Julia, and MPI, micro benchmarks, as well as scalability experiments with increasing data and cluster sizes.

2 SystemML Preliminaries

To provide the necessary background of large-scale ML systems for data-parallel operations, we first describe Apache SystemML's [Bo14; Bo16] compiler and distributed matrix representations. Subsequently, we introduce straw-man cumulative aggregates at script level and give an overview of support for cumulative aggregates in other ML and DB systems.

Script Compilation: SystemML provides a high-level language with R-like syntax to express algorithms via linear algebra such as matrix multiplications, aggregations, and statistical functions. These scripts are parsed into a hierarchy of statement blocks, delineated by control flow. For each basic block, we then compile directed acyclic graphs (DAGs) of high-level operators (HOPs) and perform various simplification rewrites. Size information is propagated from the inputs over the entire program and bottom-up through these HOP DAGs to compute memory estimates per operation. These estimates—together with driver and executor memory budgets—are in turn used to select physical operators for local and distributed operations, which eventually yield an executable runtime plan. For this purpose, we provide single-node CP (control program), as well as distributed Spark and MapReduce operators for all supported operations[4] and computation primitives.

Distributed Matrix Representation: Large-scale ML systems for data-parallel operations commonly rely on blocked matrix representations, where matrices are stored as distributed collections of block indexes and fixed-size blocks [Bo16; Lu17; Ma16; Ro17; Yu17; Za16]. Individual blocks—also known as tiles [HBY13; ZHY09] or chunks [St11]—are then stored in dense, sparse, or ultra-sparse formats. The use of squared $b \times b$ blocks (e.g., with block size $b = 1\text{K}$ in SystemML) works very well in practice because it ensures aligned blocks during join processing and aggregations independent of the join dimension, transpose operations can be computed in a block-local manner, and even dense blocks (8 MB) still fit in L3 cache, while block overheads are usually amortized across many values. For local CP operations, the entire matrix is represented as a single block to reuse runtime operations, avoid unnecessary overheads, and simplify local operations.

Straw-man Cumulative Aggregates: ML systems with language support for loops and indexing can emulate cumulative aggregates at script level. In fact, we used these alternatives in various SystemML use cases before we added built-in support in 2014. Figure1 shows two examples that both compute cumulative column sums $\mathbf{B}$ of an $n \times m$ input matrix $\mathbf{A}$. We will use these straw-man implementations for baseline comparisons. Although both functions perform only $O(nm)$ additions, they have very different characteristics and thus, it is useful to understand the resulting time complexity. First, `cumsumN2` is the most obvious implementation, where we maintain a row vector of column sums `csums` as we make a single pass over the input and output matrices. However, due to copy on write semantics, each row-wise matrix left-indexing `B[i,]=` internally copies the entire matrix $\mathbf{B}$, which results in $O(n^2m)$ time complexity. Due to the absence of loop-carried dependencies over

[4] There are few exceptions such as specific solvers and deep neural network operations for which distributed operations are rarely necessary and thus, not yet supported.

```
cumsumN2 = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; csums = matrix(0,1,ncol(A));
  for( i in 1:nrow(A) ) {
    csums = csums + A[i,];
    B[i,] = csums;
  }
}
```

```
cumsumNlogN = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A;  m = nrow(A); k = 1;
  while( k < m ) {
    B[(k+1):m,] = B[(k+1):m,] + B[1:(m-k),];
    k = 2 * k;
  }
}
```

Fig. 1: Straw-man Cumsum Functions (amenable to update-in-place).

B, this pattern qualifies for update in-place (with $O(nm)$), but only for local, in-memory operations whereas distributed RDDs are always immutable. Second, `cumsumNlogN` aims to overcome this issue by using a trick of shifted addition and exponentially increasing k, which results in a time complexity of $O(n \log n \cdot m)$ with copy on write. This shifting is similar to the data-parallel plus-scan by Hillis and Steele [HS86]. For local operations, SystemML again uses update-in-place because the data dependencies between right- and left-indexing correctly serialize the reads and writes on **B**. Finally, both script-level alternatives were unsatisfactory in practice—especially for distributed operations—which motivated the support for local and distributed built-in operations in SystemML.

Cumulative Aggregates in ML Systems: Numerical computing frameworks such as R, Matlab, and Julia all support cumsum($\mathbf{X}$), cummin($\mathbf{X}$), cummax($\mathbf{X}$), cumprod($\mathbf{X}$), while NumPy provides only cumsum($\mathbf{X}$) and cumprod($\mathbf{X}$). For handling matrix inputs, Matlab, Julia, and NumPy use overloaded functions that specify the aggregation dimension with mostly column-wise defaults. In contrast, R users explicitly apply vector operations, for example, via `apply(X, 2, cumsum)` for computing column-wise prefix sums. R also provides parallel apply functions, but they require **X** to fit in memory of a single node. In contrast, SystemML provides built-in support for local and distributed cumulative aggregates, as well as complex aggregates like cumsumprod($\mathbf{X}$), which we will discuss in Section 3.3.

Cumulative Aggregates in SQL: Given the importance in practice, we further review support for cumulative aggregates in SQL. Besides naïve approaches via self-joins and recursive formulations, the most practical approach is based on window functions as introduced in SQL:2003's OLAP operations [Me03]:

```
SELECT Rid, V, sum(V) OVER(ORDER BY Rid) AS cumsum FROM X
```

which defaults to a row range of `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. While cumulative aggregation with a running sum—as used in PostgreSQL [Le15]—works well for this scenario, Leis et al. further introduced a so-called segment tree to avoid unnecessary redundancy for semi-additive aggregation functions like min and max over variable-sized frames [Le15]. Interestingly, this segment tree is very similar to the concept of aggregation trees from the HPC literature [Bl93; CBZ90]. In contrast to these settings with ordered input data and direct access, we focus on data-parallel cumulative aggregates.

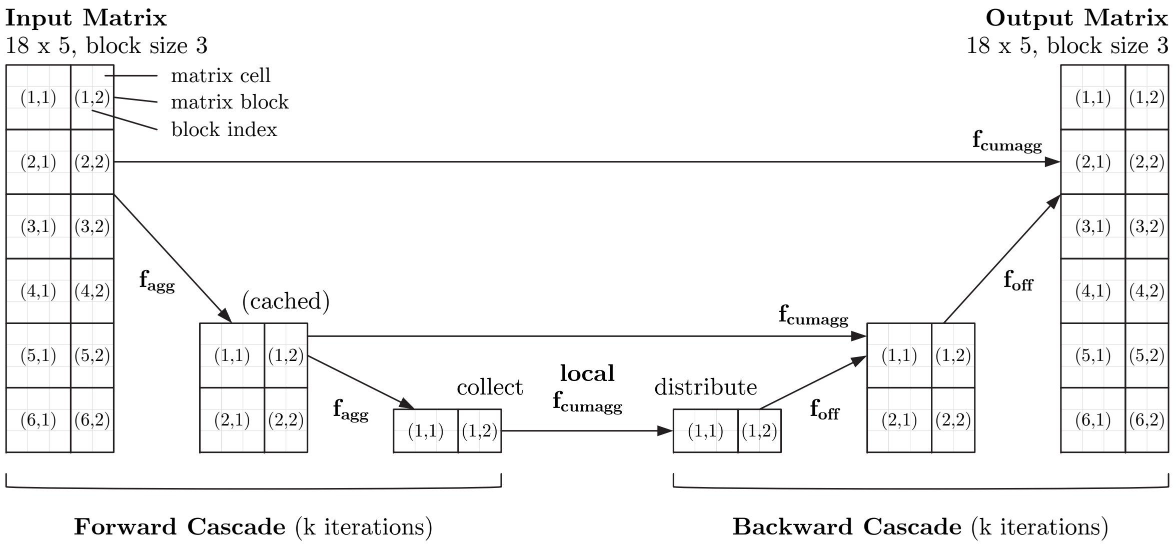


Fig. 2: Overview of the Distributed Cumulative Aggregate Framework (with k=2).

3 Data-Parallel Cumulative Aggregates

In this section, we describe DistCumAgg a framework for distributed, data-parallel cumulative aggregates. We first give an overview of the generic framework and subsequently discuss basic and complex cumulative aggregates as instantiations of this framework.

3.1 DistCumAgg Framework Overview

Framework Design: A major goal of our DistCumAgg framework is a seamless integration with blocked matrix representations, and hybrid runtime plans of local and distributed operations. At the same time, we do not make assumptions about the size of inputs or intermediates. The key idea to accomplish that is a self-similar operator chain as shown in Figure 2, consisting of a forward cascade of aggregations until the data fits into driver memory, a local cumulative aggregate over aggregates, and a backward cascade to produce the final result. Self-similarity allows for arbitrary lengths of cascades k (aggregates of aggregates) and the reuse of local cumulative aggregate operators as block operations.

Forward Cascade: Each iteration of the forward cascade applies an aggregation function f_{agg} to each matrix block to yield a row vector of column aggregates. These aggregates are then merged into an intermediate blocked matrix by mapping the input block index (i,j) to row i and column block j. Given typical block sizes of $b = 1\text{K}$, this aggregation reduces the data size by three orders of magnitude for dense input data. However, the data might still be too large. We, therefore, perform k iterations until the data fits into the driver memory. Assuming the aggregate intermediates are dense, estimating the size is simply done by scaling the input dimensions down by b^k. We do not perform these aggregations at once—as one might consider for optimization—because the intermediates are needed for the backward pass to ensure data-local operations without data shuffling.

Tab. 1: Instantiation of Basic and Complex Distributed Cumulative Aggregates.

Operation	**Init**	f_{agg}	f_{off}	f_{cumagg}
cumsum($\mathbf{X}$)	0	colSums($\mathbf{B}$)	$\mathbf{B}_{1:} = \mathbf{B}_{1:} + \mathbf{a}$	cumsum($\mathbf{B}$)
cummin($\mathbf{X}$)	∞	colMins($\mathbf{B}$)	$\mathbf{B}_{1:} = \min(\mathbf{B}_{1:}, \mathbf{a})$	cummin($\mathbf{B}$)
cummax($\mathbf{X}$)	$-\infty$	colMaxs($\mathbf{B}$)	$\mathbf{B}_{1:} = \max(\mathbf{B}_{1:}, \mathbf{a})$	cummax($\mathbf{B}$)
cumprod($\mathbf{X}$)	1	colProds($\mathbf{B}$)	$\mathbf{X}_{1:} = \mathbf{B}_{1:} \odot \mathbf{a}$	cumprod($\mathbf{B}$)
cumsumprod($\mathbf{X}$)	0	cbind(cumsumprod($\mathbf{B}$)$_{n1}$, prod($\mathbf{B}_{:2}$))	$\mathbf{B}_{11} = \mathbf{B}_{11} + \mathbf{B}_{12} \odot \mathbf{a}$	cumsumprod($\mathbf{B}$)

Local Cumulative Aggregate: The output of the last forward iteration is then collected into a local matrix at the driver, where we perform a local cumulative aggregate f_{cumagg} over the partial aggregates, yielding cumulative aggregates that can be used as offsets.

Backward Cascade: The backward cascade produces the final results based on the previously computed aggregates of all iterations and cumulative aggregate of the innermost iteration. We first parallelize (i.e., distribute) the innermost cumulative aggregate, and then perform k backward iterations inside-out. Each iteration splits the blocks of cumulative aggregates into rows and joins this collection with the corresponding input blocks of the forward cascade. The partial cumulative aggregates are then applied via f_{off} as offsets to the first row of each data block. Finally, we compute a block-local cumulative aggregate f_{cumagg} to yield the cumulative aggregate of the given iteration or final result, respectively.

Caching and Broadcasting: Regarding data-flow optimization, there are two simple techniques that greatly improved performance. First, for Spark distributed operations, we optionally cache aggregates of the forward cascade in storage level `MEMORY_AND_DISK` to avoid lazily recomputing these intermediates and thus, reading the main input multiple times. Second, to avoid unnecessary shuffling in the backward cascade, we perform a broadcast join—instead of a repartition join [Bl10]—of the data input and offsets, if the offsets fit into the broadcast memory budget. Using Spark, this applies to the innermost iteration because the broadcast variable must fit twice in memory of the driver as the `broadcast` operation serializes, compresses, and divides the in-memory object into chunks.

Instantiation: The generic DistCumAgg framework allows instantiating a variety of distributed cumulative aggregates by assigning concrete functions to f_{cumagg}, f_{agg} and f_{off}.

3.2 Basic Cumulative Aggregates

We now give an overview of instantiating the basic cumulative aggregates cumsum($\mathbf{X}$), cummin($\mathbf{X}$), cummax($\mathbf{X}$), and cumprod($\mathbf{X}$) for distributed operations. Table 1 shows the function mapping, where $\mathbf{B}$ refers to a single matrix block of $\mathbf{X}$, $\mathbf{a}$ is a row vector of cumulative aggregates joined to the data block $\mathbf{B}$, and $\odot$ denotes an element-wise multiplication. Note that the functions f_{agg}, f_{off}, f_{cumagg} reuse existing unary aggregates,

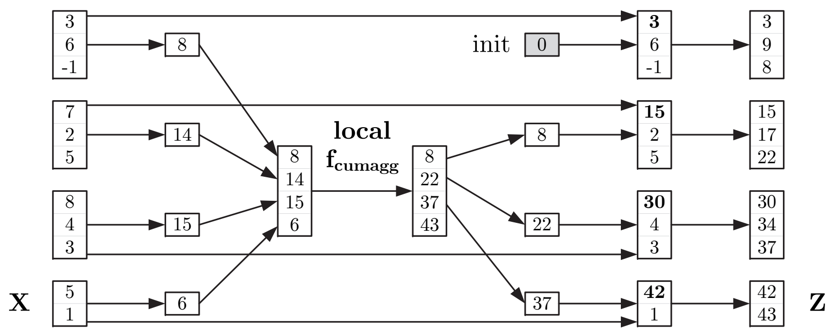


Fig. 3: Example Distributed Cumulative Sum (with k=1, $\mathbf{Z}_i = \mathbf{X}_i + \mathbf{Z}_{i-1}$).

element-wise operations, and cumulative aggregates, respectively. Since f_{off} is applied shifted by one block, we further need initialization values for the first block, which are—by operation semantics—0, ∞, $-\infty$, and 1 for cumsum, cummin, cummax, and cumprod.

Example 1 (Distributed Cumulative Sum) *Figure 3 shows a distributed cumsum with $k = 1$ iterations. We have an 11×1 input matrix, again with block size 3, resulting in 4 blocks. To compute the cumulative sum in a distributed manner, we first compute the block aggregates and merge them into a single in-memory matrix. Now we can perform cumsum($\mathbf{A}$) over the partial aggregates and distribute the intermediate. However, we need to shift these intermediates by one before the join because we want to use these as offsets for the next block. Accordingly, the first value is padded by the initialization value (0 for sum). We apply these offsets via element-wise addition to the first row of each block. Finally, we compute the block-local cumulative sum to produce the result. If we can broadcast the partial cumulative aggregates, the entire pipeline does not require any shuffle of the input matrix but only of partial aggregates, which are significantly smaller than the input.*

3.3 Complex Cumulative Aggregates

Apart from basic cumulative aggregates, SystemML also supports the complex cumulative aggregate cumsumprod that is recursively defined as follows:

$$\mathbf{Z} = \text{cumsumprod}(\mathbf{X}) = \text{cumsumprod}(\mathbf{Y}, \mathbf{W}) \quad \text{with} \quad \mathbf{Z}_i = \mathbf{Y}_i + \mathbf{W}_i \odot \mathbf{Z}_{i-1}. \tag{2}$$

Common applications of cumsumprod are weighted (e.g., decayed) cumulative sums and frame-wise cumulative sums, where any $\mathbf{W}_i = 0$ acts as a frame boundary. The interleaved additions and multiplications seem to complicate preaggregation, but because multiplication distributes over addition we can still instantiate the DistCumAgg framework as shown in Table 1. Intuitively, we factor out the scaling per block, which allows again self-similar preaggregation. First, f_{agg} now returns a 1×2 matrix computed via cbind(cumsumprod$(\mathbf{B})_{n1}$, prod$(\mathbf{B}_{:2})$). Second, f_{cumagg} remains the local cumsumprod($\mathbf{B}$) operation which produces a scalar $\mathbf{a}$ per block. And third, we reapply the scaling through f_{off} via $\mathbf{B}_{11} = \mathbf{B}_{11} + \mathbf{B}_{12} \odot \mathbf{a}$ before the final block-local cumsumprod($\mathbf{B}$).

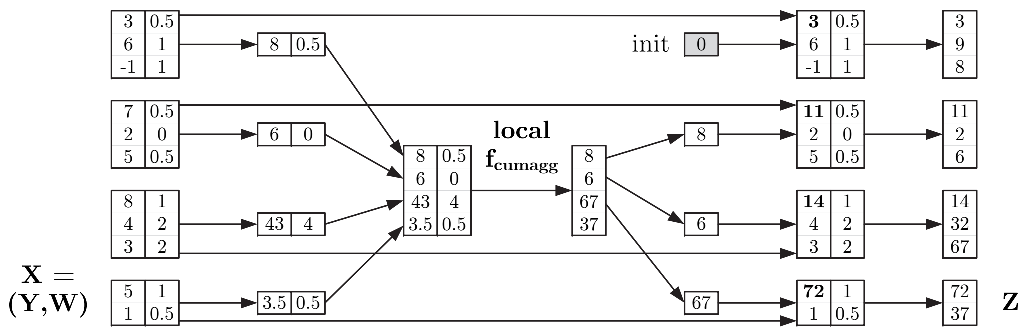


Fig. 4: Example Distributed Cumulative Sum-Product (with k=1, $\mathbf{Z}_i = \mathbf{Y}_i + \mathbf{W}_i \odot \mathbf{Z}_{i-1}$).

Data-Parallel Computation: To fit into the DistCumAgg framework for data-parallel operations, we restrict cumsumprod($\mathbf{X}$) to the special case where $\mathbf{X}$ is an $n \times 2$ matrix that concatenates the column vectors $\mathbf{Y}$ and $\mathbf{W}$. This restriction ensures that all cumulative aggregates are unary operations and that related entries in $\mathbf{Y}$ and $\mathbf{W}$ appear in the same block, which enables efficient, block-local operations.

Example 2 (Distributed Cumulative Sum-Product) *Figure 4 shows a distributed cumsumprod with $k = 1$ iterations. Similar to Example 1, the input matrix has 11 rows, block size 3, and thus, 4 blocks. The second column is the weight vector* $\mathbf{W}$*. For a distributed cumsumprod, we first compute a 1×2 aggregate per block, holding cumsumprod*$(\mathbf{B})_{n1}$ *and prod*$(\mathbf{B}_{:2})$*. These aggregates are merged into a local matrix on which we apply a cumsumprod to yield a vector of scaling factors. The factors* $\mathbf{a}$ *are then applied to the first element of respective blocks via* $\mathbf{B}_{11} = \mathbf{B}_{11} + \mathbf{B}_{12} \odot \mathbf{a}$ *for computing the final result* $\mathbf{Z}$*.*

3.4 Characterization of Applicable Operations

Following the instantiation of basic and complex cumulative aggregates, we aim to characterize the operations that are applicable to our data-parallel DistCumAgg framework.

Theoretical Foundation: The HPC literature has studied the parallel evaluation of recurrence equations extensively [Bl93; HK77]. Hence, we directly reuse known results from [Bl93] and impose additional restrictions for our DistCumAgg framework. Given a recurrence $x_i = a_i \oplus (b_i \otimes x_{i-1})$, it can be efficiently parallelized if (1) $\oplus$ is associative, (2) $\otimes$ is semi-associative (with an associative companion operator) or associative, and (3) $\otimes$ distributes over $\oplus$. These conditions apply to first- and higher-order recurrences.

Additional Restrictions: Efficient, data-parallel computation over blocked matrices, and the integration into our self-similar framework further require two restrictions:

- *No Higher-Order Recurrences:* Due to limited data access across blocks, higher-order recurrences such $x_i = a_i \oplus x_{i-1} \oplus x_{i-2}$ are disallowed. We also limit the number of inputs to less or equal the block size to ensure co-partitioning of inputs.

- *Vectorized Operations:* We require that $\oplus$ and $\otimes$ have existing aggregate and element-wise operations, which enables the composition from existing block operations.

Strength Reduction: Note that cumsumprod($\mathbf{X}$) uses cumsumprod$(\mathbf{B})_{n1}$—i.e., the last block entry—as part of f_{agg}. Similarly, for cumsum($\mathbf{X}$), we could use cumsum$(\mathbf{B})_{n:}$. However, this simplifies to colSums($\mathbf{B}$), which avoids materializing the cumsum output block.

4 System Integration

A robust and holistic system integration faces additional requirements. In this section, we discuss the end-to-end compiler and runtime integration of our DistCumAgg framework into SystemML as well as the use of cumulative aggregates in other operations.

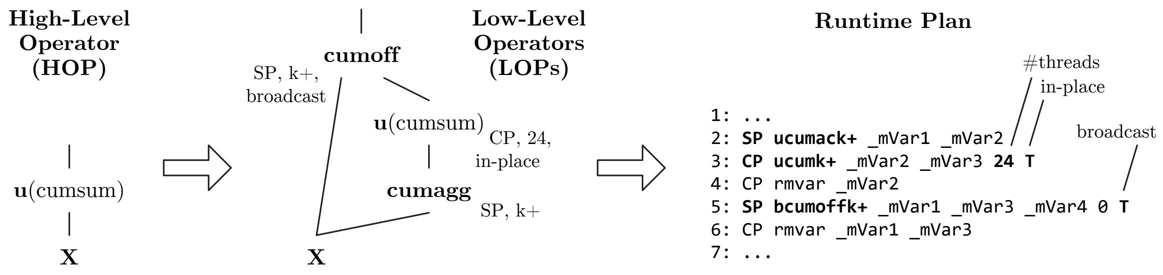
4.1 Compiler and Runtime Integration

After script parsing and validation, a cumulative aggregate is represented by a single high-level operator (HOP), specifically a typed unary operator like u(cumsum). This HOP is then compiled into plans of physical operators as follows.

Simplification Rewrites: At HOP-level, we perform multiple rounds of size propagation and simplification rewrites. First, during intra- and inter-procedural analysis, we infer the output sizes of cumulative aggregates by propagating the input dimensions and assuming dense outputs. Second, we apply the following simplification rewrites in terms of transformation rules, which avoid unnecessary intermediates and data shuffling in distributed environments:

$$\begin{aligned}
\text{rev}(\text{cumsum}(\text{rev}(\mathbf{X}))) &\rightarrow \mathbf{X} + \text{colSums}(\mathbf{X}) - \text{cumsum}(\mathbf{X}) \\
\text{colSums}(\text{cumsum}(\mathbf{X})) &\rightarrow \text{colSums}(\mathbf{X} \odot \text{seq}(\text{nrow}(\mathbf{X}), 1)) \\
\mathbf{X} \odot \text{cumsum}(\text{diag}(\text{matrix}(1, \text{nrow}(\mathbf{X}), 1))) &\rightarrow \text{lower.tri}(\mathbf{X}) \\
\text{cumsum}(\mathbf{X}) &\rightarrow \mathbf{X} \;\text{ iff }\; \text{nrow}(\mathbf{X}) = 1.
\end{aligned} \tag{3}$$

Examples of static—i.e., size-independent—rewrites are the computation of suffix sums rev(cumsum(rev($\mathbf{X}$))) from the prefix sums, as well as the computation of aggregates such as sum(cumsum($\mathbf{X}$)) or colSums(cumsum($\mathbf{X}$)) from entries in $\mathbf{X}$ scaled by their inclusion frequencies. Furthermore, the expression $\mathbf{X} \odot$ cumsum(diag(matrix(1, nrow($\mathbf{X}$), 1))) selects the lower triangular matrix of a squared matrix $\mathbf{X}$ (with nrow($\mathbf{X}$) = ncol($\mathbf{X}$)) and thus, can be extracted via lower.tri($\mathbf{X}$). In contrast, dynamic rewrites have additional size constraints: for example, cumsum($\mathbf{X}$) $\rightarrow$ $\mathbf{X}$ only applies if $\mathbf{X}$ is a single-row matrix. Finally, we also support cumulative aggregates in SystemML's code generation framework, which similarly requires size information for costing and validity constraints [Bo18].



```
...
SP ucumack+ _mVar1 _mVar2
CP ucumk+ _mVar2 _mVar3 24 T
CP rmvar _mVar2
SP bcumoffk+ _mVar1 _mVar3 _mVar4 0 T
CP rmvar _mVar1 _mVar3
...
```

Fig. 5: Compilation Chain of Cumulative Aggregates.

Execution Plan Generation: After rewrites, we then use the size information for computing memory estimates. Compared to the local memory budget, these estimates are in turn used for execution type selection of local (CP) or distributed Spark (SP) operations. A CP operation maps to a multi-threaded physical operator, while an SP operation maps to a DAG of operators as shown in Figure 5. For a single-iteration `cumsum`($\mathbf{X}$), we compile a unary SP `cumagg`, a CP cumsum, and a binary SP `cumoff`. These operators are annotated with the execution types, operation types (e.g., Kahan addition [TTR12]) and additional flags like broadcasting, number of threads, and in-place updates. This LOP DAG may also include checkpoints for caching if needed. In case of nrow($\mathbf{X}$) $\leq b$, we compile only a `cumoff` of $\mathbf{X}$ and a constant vector because the single row block does not require offsets. Finally, we generate runtime instructions from the LOP DAG in a depth-first traversal. Based on live variable analysis, this runtime plan also includes `rmvar` instructions for cleaning up temporary intermediates and their lineage-aware broadcasts and RDD variables [Bo16].

Runtime Operators: Important aspects of the runtime integration are the different physical operators and data transfer between local and distributed operations:

- *CP cumsum operator*: For local in-memory operations of f_{cumagg}, we provide a basic operator with copy-on-write semantics. This operator is parameterized with the specific f_{cumagg} and allows for single- or multi-threaded operations. In case of multi-threading, we compute offsets similar to the DistCumAgg framework but with static range partitioning in the number of threads. Optionally, this operator performs in-place updates—as shown in Figure 5—to avoid the expensive output allocation.

- *Spark Partial Cumulative Aggregate:* The data-parallel Spark `ucumac` aggregation (e.g., `ucumack+` in Figure 5) is an operator for performing f_{agg} during the forward cascade. In detail, this entails (1) a data-local block aggregation into a row of partial column aggregates, (2) the insertion of this row into its position of an empty target block, and (3) the global merge of these partial blocks. For memory efficiency and scalability, partial blocks are communicated in sparse format.

- *Spark Cumulative Offset:* The data-parallel Spark `bcumoff` offset aggregation, then applies the offsets via f_{off} and performs the final block-local f_{cumagg}. Performance is largely influenced by the join of data and offsets. If configured for broadcast, we use an efficient broadcast join that avoids data shuffling and overcomes Spark's limitation of 2GB broadcasts with partitioned broadcasts [Bo16]. Otherwise, we parallelize the offsets, split them into rows, and fall back to a repartition join. Similarly, if the data is hash partitioned, we also use a repartition join because Spark exploits the partitioning for shuffling only offsets [Bo16]. Finally, we perform a zero-copy offset aggregation during f_{cumagg} to avoid an additional copy-on-write, and thus, GC overheads.

The data transfer between CP and Spark operations is handled through SystemML's bufferpool [Bo16]. For example, each CP operation pins its inputs via `acquireRead`, which under the covers evaluates pending RDD operations and collects the output. To reduce latency, we overlap the RDD evaluation with the allocation of the local matrix.

4.2 Cumulative Aggregates in Other Operations

Cumulative aggregates are not just versatile tools at script level but also used as key primitives in other operations with complex dependencies. Examples are the removal of empty rows or columns via removeEmpty, the subsampling of rows and columns via permutation matrix multiplication, and the computation of cumulative distribution functions.

Semantics of removeEmpty: We use removeEmpty as an example. For instance, the expression `removeEmpty(target=X, margin="rows", select=rowSums(X!=0)>=thrU)` removes users—i.e., rows—with too few ratings (< thrU) from a ratings matrix $\mathbf{X}$ in preparation for matrix factorization. Generally, removeEmpty selects non-empty rows or columns, or if a `select` vector is provided all rows or columns whose vector entry is non-zero.

Data-Parallel removeEmpty: Parallelizing removeEmpty is challenging over distributed, blocked matrices because the decision on the removal of a row depends on all columns of this row, and the output position of a row depends on the removal decisions of all previous rows. However, with cumulative aggregates, parallelization becomes simple. To explain this, we decompose removeEmpty into separate phases. Local operations exploit sparse formats and early-out opportunities, but here we focus on data-parallel operations only:

- *Selection Vector Computation* (No-op if user-provided): For distributed data-parallel operations, we simply compile distributed rowMaxs($\mathbf{X} \neq 0$) or colMaxs($\mathbf{X} \neq 0$) operations, respectively. These operations result in a 0/1 indicator vector $\mathbf{s}$.
- *Row/Column Selection:* Subsequently, we want to select rows/columns from the blocked representation of $\mathbf{X}$ according to $\mathbf{s}$ into the blocked representation of the output. The key to accomplish that is a potentially distributed $\mathbf{r} = \text{cumsum}(\mathbf{s})$, where $\mathbf{r}_i$ holds the output position of the ith input row. This is similar to parallel packing of selected elements [CBZ90] and filtering through compaction [Ho05]. Finally, we simply reshuffle and merge relevant rows/columns into the target representation.

During compilation of removeEmpty, we construct a temporary HOP DAG including u(cumsum) to prepare the **r** vector. This way, we reuse the entire compiler and runtime integration described in Section 4.1 including hybrid runtime plans and internal optimizations.

5 Experiments

Our experiments study both local and distributed cumulative aggregates. First, we compare SystemML with numerical computing frameworks such as R and Julia for singlenode operations. Second, we investigate the scalability of distributed cumulative aggregates, including the impact of internal optimizations, and a comparison with MPI.

5.1 Experimental Setting

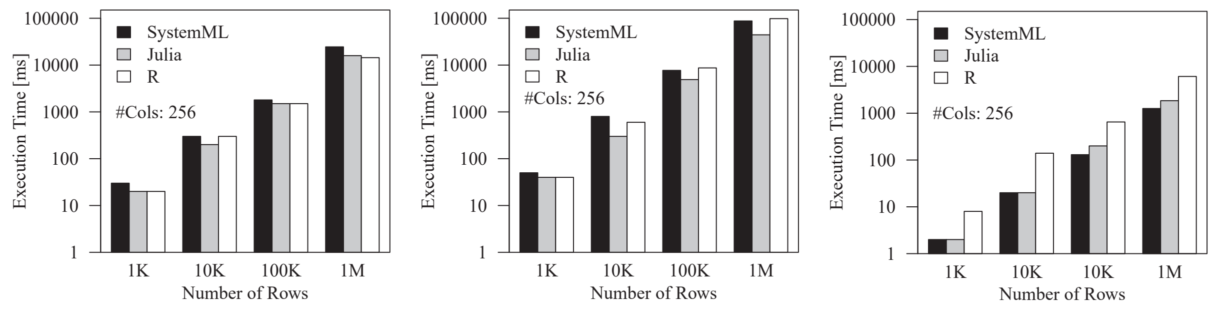
Cluster Setup: We ran our experiments on a commodity 2+10 node cluster of two head nodes, and 10 worker nodes. All nodes have two Intel Xeon E5-2620 CPUs @ 2.1 GHz-2.5 GHz (24 virtual cores), 128 GB DDR3 RAM @ 1.3 GHz, six 3 TB disks, 1Gb Ethernet, a nominal peak memory bandwidth of 2×43 GB/s, and run CentOS Linux 7.2. We used OpenJDK 1.8.0_151, Hortonworks 2.5, Apache Hadoop 2.7.3, and Apache Spark 2.3.1, in yarn-client mode, with 10 executors, 19 cores per executor, 40 GB driver memory, 60 GB executor memory, and default memory fractions (0.5/0.6), which results in an aggregate cluster memory for data between $10 \cdot 60\,\text{GB} \cdot 0.5 = 300\,\text{GB}$ and 360 GB.

Baselines and Data: Our baselines are cumsum built-in operations and the straw-man scripts from Figure 1 in SystemML 1.2++ (12/2018) [Bo16], Julia 0.7 [Be17], and R 3.5, as well as a C-based MPI baseline using OpenMPI 3.1.3. All systems use double precision (i.e., `FP64`); SystemML is configured with a default block size of b = 1K and a local memory budget of 70% of the maximum heap size. Finally, we use synthetic data for studying these baselines on a variety of scenarios with different data sizes.

5.2 Baseline Comparison

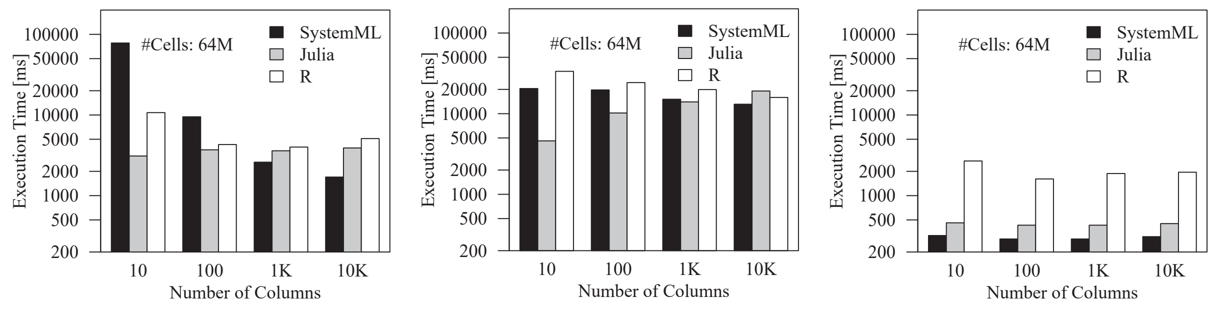
In a first set of experiments, we compare the script-level straw-man implementations and built-in support in SystemML, R, and Julia to provide a baseline for distributed operations. We use a single worker node, with `-Xmx96g -Xms96g -server` for SystemML, unlimited memory for R and Julia, and we report the average runtime of 100 operations.

Increasing Data Size: For a basic comparison, we fix the number of columns as $m = 256$ and systematically increase the number of rows. Figure 6 shows the results with log-scaled



(a) Script cumsumN2 (b) Script cumsumNlogN (c) Built-in cumsum

Fig. 6: Cumsum Performance with Increasing Number of Rows, Constant Number of Columns.



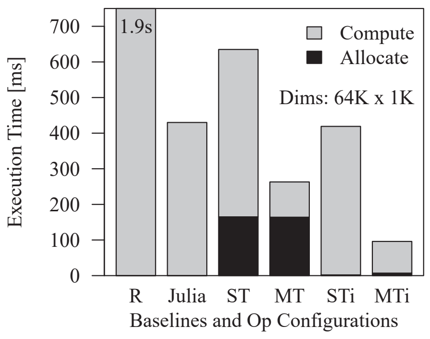
(a) Script cumsumN2 (b) Script cumsumNlogN (c) Built-in cumsum

Fig. 7: Cumsum Performance with Increasing Number of Columns, Decreasing Number of Rows.

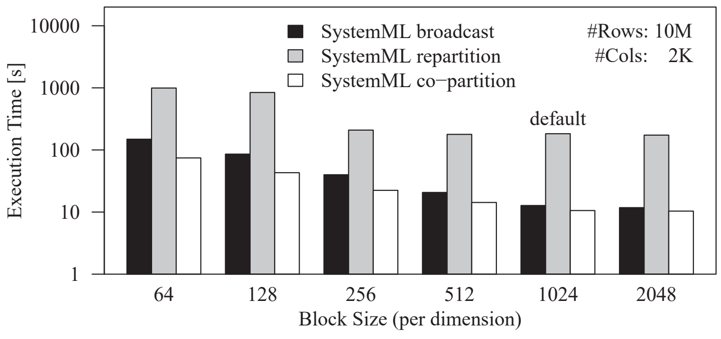
axes. First, although cumsumN2 and cumsumNlogN are amenable to update-in-place[5], these script level functions are one to two orders of magnitude slower than the built-in functions. Here, cumsumNlogN shows higher overhead than cumsumN2 due to larger intermediates. However, without update-in-place rewrite in SystemML, cumsumNlogN largely outperforms cumsumN2 (e.g., 0.9 s vs. 86.7 s for 10K rows). Second, Julia performs best on cumsumN2 and cumsumNlogN because for SystemML and R, instruction interpretation overhead constitutes a bottleneck at $n = 256$ columns. Third and most importantly, the built-in functions of SystemML, and Julia show very similar performance, while R has additional overhead for its column-wise `apply`. SystemML slightly outperforms Julia due to multi-threaded operations, which are, however, dominated by result allocation.

Varying Matrix Shape: To validate the observation of high instruction interpretation overheads, Figure 7 shows the results for a constant data size of 64M cells (512 MB) but increasing number of columns (and thus, decreasing number of rows). First, Figure 7(c) shows that all built-in functions are rather invariant to the number of columns. Second, Figures 7(a) and 7(b) show that SystemML's and R's performance increases with increasing number of columns and thus decreasing number of operations, which is especially pronounced for cumsumN2 (Figure 7(a)) where the number of operations is linear in the number of rows. In

[5] SystemML and R have copy-on-write semantics but apply update-in-place via rewrites and reference counting. In contrast, Julia uses update-in-place by default and requires an explicit `B = copy(A)` if this is not desired.



(a) In-Place Operations (512 MB)



(b) Impact of Broadcasting and Block Sizes

Fig. 8: Impact of Multi-Threading, In-place Updates, Broadcasting and Block Sizes.

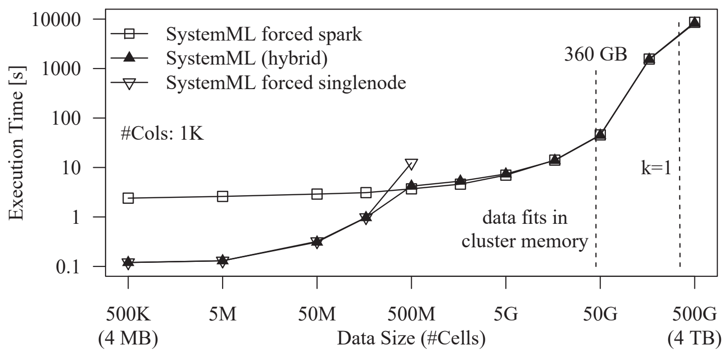
fact, SystemML outperforms R and Julia for these script-level algorithms if the number of columns is sufficiently large. Surprisingly, Julia shows a slowdown with increasing number of columns, especially for cumsumNlogN, which is likely due to GC overheads.

In-Place Operations: For the sake of a better understanding of builtin operations performance, Figure8(a) shows the impact of multi-threading and in-place updates, as well as a break-down of computation times. Here, ST/MT stand for single- and multi-threading, while the i-suffix indicates in-place updates. Multi-threading improves the compute time by 4.7x, but its runtime is then dominated by result allocation. Hence, in-place operations—as applied between forward and backward cascade—further significantly improve performance.

5.3 Impact of Broadcasting and Block Sizes

In a second set of experiments, we evaluate distributed cumulative aggregates in SystemML for the common scenario of dense matrices that fit in aggregate cluster memory and require only $k = 1$ iterations. We use a—randomly generated—dense 10M × 2K input matrix (i.e., 160 GB) and study the impact of broadcasting, partitioning, and block sizes. To force Spark's lazy evaluation, we report the average runtime of 100 `print(min(cumsum(X)))` evaluations, including creating the Spark context once ($\approx$ 15 s).

Broadcast vs Repartition Join: As described in Section 4.1, joining the data and computed offsets via a broadcast or co-partition join instead of a repartition join, allows for data-local computation without shuffling of the main input. Broadcast joins apply only to the innermost iteration, while co-partition joins apply to all iterations, except the outermost iteration unless the data is already partitioned. Here, we force the individual types and repartition the data for co-partition joins just once. Figure 8(b) shows significant improvements with broadcast/co-partition joins—whose differences will be analyzed separately—compared to a repartition join by up to $19.6x$ ($17.3x$ at default block size of $b = 1\text{K}$).



(a) SystemML End-to-End Performance

#Cells	SystemML	MPI
165M	0.97 s	0.14 s
500M	4.2 s	0.26 s
1.65G	5.3 s	0.61 s
5G	7.4 s	1.96 s
16.5G	13.9 s	6.20 s
50G	44.8 s	19.8 s
165G	1,531 s	N/A
500G	8,291 s	N/A

(b) `MPI_Scan` Baseline

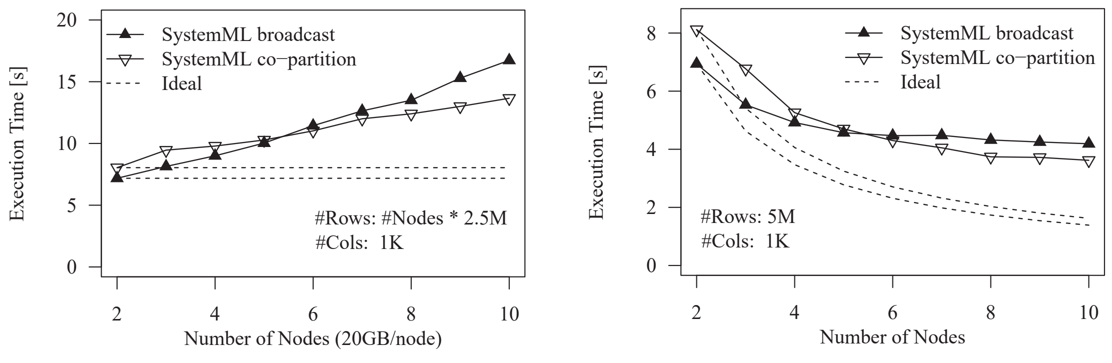
Fig. 9: Scalability with Increasing Data Size (from 4 MB to 4 TB).

Impact of Block Sizes: In SystemML, we use 1K as the default block size because it offers a great balance between (1) low per-block memory requirements and good cache behavior, as well as (2) amortized block overheads. For distributed cumulative aggregates, the block size also influences the size reduction per iteration, and thus, the number of partial aggregates, the number of iterations, and the broadcast overhead. In Figure 8(b), we observe—despite the moderate range of block sizes from 64 to 2048—significant performance impact of up to more than an order of magnitude. Repartition joins are less sensitive to block sizes because they are dominated by shuffling, but the default of 1K is generally a good compromise.

5.4 Scalability

In a third set of experiments, we test the scalability of cumulative aggregates over a spectrum of data and cluster sizes. We use a 10 GB driver and report the average runtime of r `print(min(cumsum(X)))` evaluations, including creating the Spark context once.

From 4 MB to 4 TB: Figure 9(a) shows the runtime of SystemML's execution modes—with 10 GB driver and $r = 10$ repetitions—with increasing data size. Forced single-node operations are fast for small data but run out-of-memory at 40 GB. Already at 4 GB, we see a slowdown because pinning 2×4 GB in a 10 GB driver leads to evictions on every cumsum. In contrast, forced Spark operations show high overhead for small data due to Spark context creation, low parallelism in the number of partitions, and distribution overheads, but good performance for distributed, in-memory data and scalability for larger datasets. When the data size exceeds cluster memory at $>$ 50G cells, we see a 34.2x slowdown for 3.3x increase in data size. Similarly, multi-iteration, repartition-based operations cause another 5.4x slowdown for the next 3.3x size increase. SystemML's default hybrid mode combines the advantages and provides efficiency for small as well as scalability for large datasets. Table 9(b) further shows an idealized MPI baseline (implemented in C) using 10 nodes and 19 slots per node. This baseline performs worker-local data generation, as well as $r = 10$ repetitions of a local f_{agg}, `MPI_Scan` (`MPI_SUM`), local f_{cumoff}, f_{cumagg}, and aggregation, as



(a) Weak Scaling (WS) (b) Strong Scaling (SS)

Fig. 10: Scalability with Increasing Cluster Size (from 2 to 10).

well as a final `MPI_Reduce` (`MPI_MIN`). For sufficiently large data, SystemML shows—despite the handling of unordered blocks—only a 2.3x slowdown and thus, competitive performance.

Weak and Strong Scaling: Finally, we study the weak and strong scaling behavior of broadcast- and co-partition-based cumulative aggregates with $r = 100$ repetitions. First, for weak scaling (WS), we simultaneously increase the data and cluster size and thus, expect constant runtime. Figure 10(a) shows moderately increasing runtimes but generally good scaling behavior. Specifically, we have a 2.3x and 1.7x slowdown when increasing from (2 nodes/40 GB) to (10 nodes/200 GB). Second, for strong scaling (SS), we increase the cluster size but keep the data size constant at 40 GB. Comparing the runtime with 2 and 10 nodes, we expect a speedup of $5x$ but observe only $1.7x$ and $2.3x$. This is due to the small data size of 40 GB, a moderate number of $\approx 40\,\text{GB}/128\,\text{MB} = 312$ partitions (and thus, tasks), and a decreasing number of task waves $\lceil 312/(19 \cdot \#\text{Nodes}) \rceil$. Overall, co-partitioning scales better because the initial partitioning is amortized over 100 repetitions, and piecewise broadcast fetching shows high latency over 1Gb Ethernet. To validate this explanation, we repeated the broadcast experiments on a 6 node cluster of similar HW but 10Gb Ethernet. When scaling from 2 to 6 nodes, we see a significantly better WS runtime overhead of 1.3x instead of 1.6x, and an SS speedup of 2.3x instead of 1.5x.

6 Related Work

We review related work from high-performance computing (HPC), database systems, recent large-scale ML systems, and the more general problem of sum-product optimization.

HPC Parallel Prefix Scans: Parallel prefix sums are a key building block in many HPC applications and therefore well-studied in the literature. Hillis and Steele presented an $O(n \log n)$ parallel scan[6] (prefix) operation [HS86] for computing prefix sums—similar to `cumsumNlogN` in Figure 1—that generalized to associative operations. Blelloch later

[6] The name *scan* was derived from the APL [FI73] plus-scan for computing vector prefix sums [Bl93].

publicized more efficient parallel prefix scans, where for n data items and p processors, each processor sums up $\lceil n/p \rceil$ elements, and partial offsets are computed via an up- and down-sweep through a virtual aggregation tree [Bl89; Bl93; CBZ90]. These scan primitives are widely applicable to evaluate polynomials, solve recurrences or tridiagonal linear systems, and even implement radix- and quick-sort [CBZ90]. Sanders further analyzed and compared alternatives for `MPI_Scan` [ST06]. Parallel algorithms for first- and higher-order recurrences have also been theoretically investigated [CK75; HK77; KMW67; KS73]. More recently, several works introduced efficient scan primitives for GPU [Ho05; HSO07] and FPGA [AS14] devices. However, all of these works focused on traditional *message-passing* or *shared-memory* HPC systems with random data access. In contrast, we described efficient cumulative aggregates for data-parallel ML systems with blocked matrix representations on top of frameworks such as MapReduce [DG04], Spark [Za12], or Flink [Al14].

Window Functions in Database Systems: Recently, the parallelization of cumulative aggregates has also received attention in the context of efficiently evaluating SQL window functions [Be13; Le15]. Database systems primarily parallelize over independent partitions of the `PARTITION BY` clause, but additional approaches exist for large partitions. First, Bellamkonda et al. introduced a method [Be13] that includes the `ORDER BY` keys into Oracle's data distribution keys to create artificial partitions. Based on an existing range partitioning and sorting, the query coordinator then collects partial aggregates and distributes offsets back. Second, Leis et al. introduced the dynamic classification of partitions for inter- and intra-partition parallelism [Le15], where the latter relies on parallel sorting, and materializing a segment tree of partial aggregates, which can be computed in parallel. In contrast to our DistCumAgg framework, these works rely on partitioning and sorting, focus primarily on shared memory, and do not support complex cumulative aggregates.

Large-Scale ML Systems: Predominant architectures for large-scale—i.e., distributed—ML Systems are (1) data- or model-parallel parameter servers [De12; Li14], and (2) systems for data- and/or task-parallel distributed operations. Distributed cumulative aggregates are only relevant for the second category of data-parallel ML systems. In this context, systems such as RIOT [ZHY09], PEGASUS [KTF09], SystemML [Bo16], SciDB [St11], Cumulon [HBY13], Distributed R [Ma16], DMac [YSC15], Spark MLlib's block matrices [Za16], Gilbert [Ro17], MatFast [Yu17], and SimSQL [Lu17] all rely on blocked matrix representations, which means that our DistCumAgg framework could seamlessly be applied in these systems. Yet, to the best of our knowledge, SystemML is the only ML system that provides built-in support for distributed cumulative aggregates.

Sum-Product Optimization: In contrast to most ML systems like OptiML [Su11], Theano [Be10], SystemML [Bo16], and TensorFlow XLA [Go] that perform simplification rewrites via heuristic pattern matching, sum-product optimization aims at a systematic exploration of rewrite opportunities. Existing work such as AMF [MP99] and SystemML-SPOOF [El17] use axioms and algebraic properties, or elementary transformation rewrites to systematically explore valid, alternative plans. Due to the complexity of reasoning about recurrences, none of these sum-product frameworks support cumulative aggregates yet. However, it is an interesting direction for future work to automatically derive rewrites for these complex cumulative aggregates, especially regarding the interactions with other operations.

7 Conclusions

To summarize, we introduced the generic DistCumAgg framework for efficient, data-parallel cumulative aggregates over distributed, blocked matrix representations. We described the end-to-end compiler and runtime integration in SystemML, and physical operators that seamlessly fit into hybrid runtime plans of local and distributed operations. Our experiments have shown competitive performance for local and distributed in-memory operations, as well as almost linear scalability with increasing data size and degree of parallelism. In conclusion, many presumably sequential operations can be efficiently computed over distributed, blocked matrix representations on top of data-parallel frameworks like Spark. Cumulative aggregates are used in a variety of algorithms and for data preprocessing, which makes them valuable tools for data scientists. Interesting future work includes sum-product rewrites, dedicated GPU operators, and other recursive computations like time series analysis.

Acknowledgments: We thank José Molero for help with the MPI experiments and our anonymous reviewers for their valuable comments and suggestions.

References

[Al14] Alexandrov, A. et al.: The Stratosphere platform for big data analytics. VLDB J. 23/6, 2014.

[AS14] Arap, O.; Swany, M.: Offloading MPI Parallel Prefix Scan (MPI_Scan) with the NetFPGA. CoRR abs/1408.4939/, 2014.

[Be10] Bergstra, J. et al.: Theano: a CPU and GPU Math Expression Compiler. In: SciPy. 2010.

[Be13] Bellamkonda, S. et al.: Adaptive and Big Data Scale Parallel Execution in Oracle. PVLDB 6/11, 2013.

[Be17] Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V. B.: Julia: A Fresh Approach to Numerical Computing. SIAM Review 59/1, 2017.

[Bl10] Blanas, S. et al.: A comparison of join algorithms for log processing in MapReduce. In: SIGMOD. 2010.

[Bl89] Blelloch, G. E.: Scans as Primitive Parallel Operations. IEEE Trans. Computers 38/11, 1989.

[Bl93] Blelloch, G. E.: Prefix Sums and Their Applications, tech. rep., CMU-CS-90-190, https://www.cs.cmu.edu/ guyb/papers/Ble93.pdf, 1993.

[Bo14] Boehm, M. et al.: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. IEEE Data Eng. Bull. 37/3, 2014.

[Bo16] Boehm, M. et al.: SystemML: Declarative Machine Learning on Spark. PVLDB 9/13, 2016.

[Bo18] Boehm, M. et al.: On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. PVLDB 11/12, 2018.

[CAL94] Cohn, D. A.; Atlas, L. E.; Ladner, R. E.: Improving Generalization with Active Learning. Machine Learning 15/2, 1994.

[CBZ90] Chatterjee, S.; Blelloch, G. E.; Zagha, M.: Scan primitives for vector computers. In: SC. 1990.

[CGJ94] Cohn, D. A.; Ghahramani, Z.; Jordan, M. I.: Active Learning with Statistical Models. In: NIPS. 1994.

[CK75] Chen, S.; Kuck, D. J.: Time and Parallel Processor Bounds for Linear Recurrence Systems. IEEE Trans. Computers 24/7, 1975.

[Co09] Cohen, J.; Dolan, B.; Dunlap, M.; Hellerstein, J. M.; Welton, C.: MAD Skills: New Analysis Practices for Big Data. PVLDB 2/2, 2009.

[Da10] Das, S. et al.: Ricardo: integrating R and Hadoop. In: SIGMOD. 2010.

[De12] Dean, J. et al.: Large Scale Distributed Deep Networks. In: NIPS. 2012.

[DG04] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI. 2004.

[El17] Elgamal, T. et al.: SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In: CIDR. 2017.

[FI73] Falkoff, A. D.; Iverson, K. E.: The Design of APL. IBM Journal of Research and Development 17/5, 1973.

[Gh11] Ghoting, A. et al.: SystemML: Declarative machine learning on MapReduce. In: ICDE. 2011.

[Go] Google: TensorFlow XLA (Accelerated Linear Algebra), tensorflow.org/performance/xla/.

[HBY13] Huang, B.; Babu, S.; Yang, J.: Cumulon: Optimizing Statistical Data Analysis in the Cloud. In: SIGMOD. 2013.

[HK77] Hyafil, L.; Kung, H. T.: The Complexity of Parallel Evaluation of Linear Recurrences. J. ACM 24/3, 1977.

[Ho05] Horn, D.: Stream Reduction Operations for GPGPU Applications. In: GPU Gems 2. 2005.

[HS86] Hillis, W. D.; Steele Jr., G. L.: Data Parallel Algorithms. Commun. ACM 29/12, 1986.

[HSO07] Harris, M.; Sengupta, S.; Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA. In: GPU Gems 3. 2007.

[KMW67] Karp, R. M.; Miller, R. E.; Winograd, S.: The Organization of Computations for Uniform Recurrence Equations. J. ACM 14/3, 1967.

[KS73] Kogge, P. M.; Stone, H. S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. IEEE Trans. Computers 22/8, 1973.

[KTF09] Kang, U.; Tsourakakis, C. E.; Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System. In: ICDM. 2009.

[Le15] Leis, V.; Kundhikanjana, K.; Kemper, A.; Neumann, T.: Efficient Processing of Window Functions in Analytical SQL Queries. PVLDB 8/10, 2015.

[Li14] Li, M. et al.: Scaling Distributed Machine Learning with the Parameter Server. In: OSDI. 2014.

[Lu17] Luo, S.; Gao, Z. J.; Gubanov, M. N.; Perez, L. L.; Jermaine, C. M.: Scalable Linear Algebra on a Relational Database System. In: ICDE. 2017.

[Ma16] Ma, E.; Gupta, V.; Hsu, M.; Roy, I.: dmapply: A Functional Primitive to Express Distributed Machine Learning Algorithms in R. PVLDB 9/13, 2016.

[Me03] Melton, J.: ISO-ANSI Working Draft SQL/Foundation, 2003.

[MP99] Menon, V.; Pingali, K.: High-Level Semantic Optimization of Numerical Codes. In: ICS. 1999.

[Ra16] Ratner, A. J.; Sa, C. D.; Wu, S.; Selsam, D.; Ré, C.: Data Programming: Creating Large Training Sets, Quickly. In: NIPS. 2016.

[RMJ06] Raghavan, H.; Madani, O.; Jones, R.: Active Learning with Feedback on Features and Instances. Journal of Machine Learning Research 7/, 2006.

[Ro17] Rohrmann, T.; Schelter, S.; Rabl, T.; Markl, V.: Gilbert: Declarative Sparse Linear Algebra on Massively Parallel Dataflow Systems. In: BTW. 2017.

[Sc15] Schelter, S. et al.: Efficient Sample Generation for Scalable Meta Learning. In: ICDE. 2015.

[ST06] Sanders, P.; Träff, J. L.: Parallel Prefix (Scan) Algorithms for MPI. In: PVM/MPI. 2006.

[St11] Stonebraker, M.; Brown, P.; Poliakov, A.; Raman, S.: The Architecture of SciDB. In: SSDBM. 2011.

[Su11] Sujeeth, A. K. et al.: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In: ICML. 2011.

[Tr18] Tremblay, J. et al.: Training Deep Networks With Synthetic Data: Bridging the Reality Gap by Domain Randomization. In: CVPR Workshops. 2018.

[TTR12] Tian, Y.; Tatikonda, S.; Reinwald, B.: Scalable and Numerically Stable Descriptive Statistics in SystemML. In: ICDE. 2012.

[YSC15] Yu, L.; Shao, Y.; Cui, B.: Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In: SIGMOD. 2015.

[Yu17] Yu, Y. et al.: In-Memory Distributed Matrix Computation Processing and Optimization. In: ICDE. 2017.

[Za12] Zaharia, M. et al.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: NSDI. 2012.

[Za16] Zadeh, R. B. et al.: Matrix Computations and Optimization in Apache Spark. In: KDD. 2016.

[ZHY09] Zhang, Y.; Herodotou, H.; Yang, J.: RIOT: I/O-Efficient Numerical Computing without SQL. In: CIDR. 2009.