Hyper Dimension Shuffle: Efficient Data Repartition at Petabyte Scale in SCOPE

Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, Jaliya Ekanayake
Microsoft Corporation
{shiqiao, adnico, jinsu, marcfr, hirenp, jaliyaek}@microsoft.com

ABSTRACT

In distributed query processing, data shuffle is one of the most costly operations. We examined scaling limitations to data shuffle that current systems and the research literature do not solve. As the number of input and output partitions increases, naïve shuffling will result in high *fan-out* and *fan-in*. There are practical limits to *fan-out*, as a consequence of limits on memory buffers, network ports and I/O handles. There are practical limits to *fan-in* because it multiplies the communication errors due to faults in commodity clusters impeding progress. Existing solutions that limit *fan-out* and *fan-in* do so at the cost of scaling quadratically in the number of nodes in the data flow graph. This dominates the costs of shuffling large datasets.

We propose a novel algorithm called Hyper Dimension Shuffle that we have introduced in production in SCOPE, Microsoft's internal big data analytics system. Hyper Dimension Shuffle is inspired by the divide and conquer concept, and utilizes a recursive partitioner with intermediate aggregations. It yields *quasilinear* complexity of the shuffling graph with tight guarantees on *fan-out* and *fan-in*. We demonstrate how it avoids the shuffling graph blow-up of previous algorithms to shuffle at petabyte-scale efficiently on both synthetic benchmarks and real applications.

PVLDB Reference Format:
Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, Jaliya Ekanayake. Hyper Dimension Shuffle: Efficient Data Repartition at Petabyte Scale in Scope. *PVLDB*, 12(10): 1113-1125, 2019.
DOI: https://doi.org/10.14778/3339490.3339495

1. INTRODUCTION

Today distributed relational query processing is practiced on ever larger datasets often residing on clusters of commodity servers. It involves partitioned storage of data at rest, partitioned processing of that data on workers, and data movement operations or shuffles [1, 15, 17, 21, 26]. A dataset $S = \{s_1, \ldots, s_p\}$ is horizontally partitioned into p files of rows. There is a partition function to assign rows to partitions, implementing for example a hash or range partitioning scheme. A dataset at rest with partitioning S may be read and processed by an operation with a partitioning requirement that is not satisfied by S. For example, a group-by-aggregation operator grouping on columns C requires input data partitioned on some subset of C. When that happens, the data must be shuffled from S into a new partitioning T that satisfies that requirement, such as $hash(C, 100)$.

Data shuffle is one of the most resource-intensive operations in distributed query processing [1, 14, 26]. Based on the access patterns of the SCOPE system's workload of data processing jobs at Microsoft, it is the third most frequently used operation and the most expensive operator overall. Thus, the implementation of the data shuffle operator has a considerable impact on the scalability, performance, and reliability of large scale data processing applications.

A full shuffle needs to move rows from everywhere to everywhere – a complete bipartite graph from sources to destinations with a quadratic number of edges. Using a naïve implementation of the full shuffle graph, large datasets experience high *fan-out* and high *fan-in*. A single partitioner has data for all recipients. High *fan-out* of the partitioner results in a bottleneck at the source due to high memory buffer requirements and excessive random I/O operations. Meanwhile, the mergers have to read data from many sources. High *fan-in* means numerous open network file handles and high communication latencies. In addition, high *fan-in* blocks forward progress because the multiplication of the probability of connection failures eventually requires the vertex to give up and restart. We see this occur regularly in practice. The *fan-out* and *fan-in* issues limit the scale of data shuffle operations and therefore the scalability of data processing in general. Given a hardware and OS configuration, the maximum *fan-out* and *fan-in* of shuffle should be considered as givens.

To address the scalability challenges of data shuffle operations, we propose a new data shuffle algorithm called Hyper Dimension Shuffle (abbreviated as HD shuffle), which yields *quasilinear* complexity of the shuffling graph while guaranteeing fixed maximum *fan-out* and *fan-in*. It partitions and aggregates in multiple iterations. By factoring partition keys into multi-dimensional arrays, and processing a dimension each iteration, it controls the *fan-out* and *fan-in* of each node in the graph. Since data shuffle is fundamental to distributed query processing [15, 17, 21, 26], the HD shuffle algorithm is widely applicable to many systems to improve data shuffle performance irrespective of system implementation details. Our contributions in this paper can be summarized as follows:

1. We describe HD shuffle algorithm with its primitives: reshapers, hyper partitioners, and dimension mergers. We prove the *quasilinear* complexity of the algorithm given the maximum *fan-out* and *fan-in*.

2. We implement HD shuffle in Microsoft's internal distributed query processing system, SCOPE. Several optimizations are

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339495

also proposed to better accommodate HD shuffle. Through the benchmark experiments and real workload applications, we demonstrate significant performance improvements using HD shuffle in SCOPE. In a first for published results we know of, we demonstrate petabyte scale shuffles of benchmark and production data.

3. To show the generality of HD shuffle algorithm, we also implement it in Spark. The benchmark results presented show HD shuffle outperforming the default shuffle algorithm in Spark for large numbers of partitions.

The rest of the paper is organized as follows. In Section 2, we describe the state of the art in data shuffle. In Section 3, we describe the details of Hyper Dimension Shuffle algorithm, prove its *quasilinear* complexity and demonstrate how the algorithm functions with an example. Section 4 describes the implementation of HD shuffle in SCOPE, including the execution model, specific optimizations and streaming compatibility. In Section 5, we present the performance analysis of HD shuffle algorithm by comparing with the state of the art systems on both synthetic benchmarks and real world applications in SCOPE and Spark. Section 6 provides an overview of related work that attempts to address the scalability challenges of data shuffle. Finally, we conclude in Section 7.

2. BACKGROUND

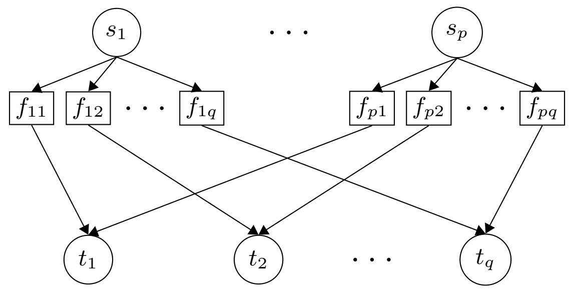


Figure 1: Shuffling data from p source partitions to q target partitions in MapReduce

MapReduce [17] formalized the problem of shuffling rows from $S = \{s_1, \ldots, s_p\}$ source partitions to $T = \{t_1, \ldots, t_q\}$ target partitions. Figure 1 shows a shuffling graph G of tasks and their intermediate files. Each partitioner task $s_i \in S$ creates one intermediate file for each target, $\{f_{i1}, \ldots, f_{iq}\}$. Each aggregator $t_j \in T$ reads one file from each source, $\{f_{1j}, \ldots, f_{pj}\}$. The data flow from S to T is a complete bipartite graph $G = (S, T, S \times T)$. The size complexity of G is dominated by the number of edges, $|S| \times |T|$. Dryad [21] and Tez [3] generalize MapReduce to general data-flow programs represented as directed acyclic graphs (DAGs) in which nodes are arbitrary data processing operations, and edges are data movement *channels*. Edges encode shuffles and various other flavors of data movement like joins and splits. There are various systems that use this graph data-centric model, that vary in data processing primitives, programming and extensibility models, and streaming vs. blocking channel implementations. Shuffling is common to all of them.

2.1 Scaling the Number of Target Partitions

The *fan-out* of a vertex $s_i \in S$ grows linearly with the number of target partitions $|T|$. The partitioning tasks in figure 1 write the intermediate files f_{ij} to disk. As tuples can arrive in any random order, an output buffer is allocated in memory for every partition to group tuples into an efficient size to write. Since the memory in the system is fixed, as $|T|$ grows to swamp the buffer memory, any scheme to free up or shrink buffers must result in more frequent, smaller, and more expensive I/O writes. And since there are many files to interleave writes to, random seeking increases.

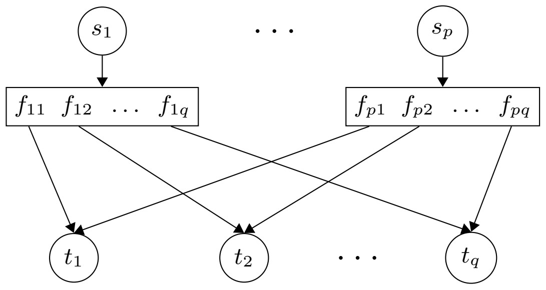

s_1
$\cdots$
s_p
f_{11} f_{12} $\ldots$ f_{1q}
f_{p1} f_{p2} $\ldots$ f_{pq}
t_1
t_2
$\cdots$
t_q

Figure 2: Writing a single sorted indexed file per partitioning task

SCOPE [13], Hadoop [1] and Spark [2, 10] use a variant of the partitioner tasks to produce a single file f_i, independent of the size of $|T|$, as shown in figure 2. To achieve this, the tuples belonging to each source partition are first sorted by the partitioning key, and then written out to the file f_i in the sorted order, together with an index that specifies the boundaries of each internal partition f_{ij} belonging to target task t_j. This removes the small disk I/Os altogether, at the cost of adding an expensive and *blocking operator* – sort – into the partitioners. Thus, an aggregator can not start reading until the partitioner writes the complete sorted file, as a total order can only be created after every tuple is processed in the source partition. This introduces a bottleneck, particularly for streaming scenarios, where the data continuously arrives and pipelines of operators can be strung together with streaming channels to provide line rate data processing.

Port exhaustion happens if the aggregators reading the same indexed file run concurrently. Then up to q network connections to a single machine are attempted, as intermediate files are typically [13, 17] written locally by the task that produced them. If the size of each partition is fixed at `1GB`, a `1PB` file requires $|T| = 10^6$ target partitions, which is above the number of ports that can be used in TCP/IP connections.

High connection overhead becomes an issue when the amount of data traveling on an edge of G becomes too small. Counterintuitively, because of the quadratic number of edges, if partition size is held constant, increasing the amount of data shuffled decreases the amount that travels along each edge! Consider a source partition having size `1GB` and the number of target partitions to be $|T| = 10^3$. Assuming a uniform distribution of the target keys, each f_{ij} would then yield around `1MB` worth of data. This payload can decrease to `1KB` as we grow to $|T| = 10^6$. Overheads of tiny connections start to dominate job latency.

2.2 Scaling the Number of Source Partitions

The *fan-in* of a vertex $t_j \in T$ grows linearly with the number of source partitions $|S|$. Memory pressure due to the number of input buffers and port exhaustion are thus also issues on the aggregator side.

Hadoop and Spark aggregators limit the number of concurrent network connections and the amount of buffer memory used. While reading a wave of inputs, if buffer memory is exhausted they spill an aggregated intermediate result to disk, which is then reread in

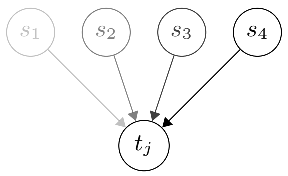


Figure 3: Aggregation with temporal order matching shading

a later wave. The temporal separation this creates is shown in figure 3.

Failure is ever-present in a large commodity cluster. Individual connections fail for a variety of reasons, some transient and some persistent. We model commodity clusters as having a low independent probability P_f of point-to-point connection failure. The probability of failure for an aggregator increases exponentially with the number of input connections, since if one of the $|S|$ intermediate files, say f_{4j}, fails to be read by target t_j, then all the aggregating progress of t_j is lost and t_j is rerun. The probability of success is $(1 - P_f)^{\textit{fan-in}}$, so increasing *fan-in* reduces the chance of progress. For a given value of P_f, the probability of success quickly switches from nearly 1 to nearly 0 at some point as *fan-in* increases.

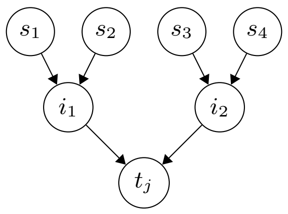


Figure 4: Aggregation with maximum fan-in $\delta_{in} = 2$

SCOPE uses a limit δ_{in} on the fan-in of any vertex. When $\delta_{in} < |S|$, a δ_{in}-ary tree is constructed to aggregate, with intermediate vertices in between the target t_j and the sources, as shown in figure 4. This helps make progress in the presence of failure in two ways: by decreasing the chance of aggregator node failure, and by decreasing the unit of work to rerun in case of failure. For example, a failure of task i_2 in reading the file written by s_4 will not cause the aggregating progress of i_1 to be lost.

The *scheduling overhead*, however, now amplifies the connection overhead. The number of intermediate tasks is $|S| \times |T|$ assuming δ_{in} is constant, as one tree is constructed for each target $t_j \in T$. The quadratic number of intermediate tasks dominates the complexity of the shuffling graph.

Riffle [28] performs machine-level merging on the intermediate partitioned data in the partitioners which converts the larger number of small, random shuffle I/O requests in Spark into fewer large, sequential I/O requests. This can be viewed as a variant of aggregation trees.

3. HYPER DIMENSION SHUFFLE

We set out to solve the following data shuffling problem. Build a data flow graph G to shuffle a rowset $S = \{s_1, \ldots, s_p\}$ into $T = \{t_1, \ldots, t_q\}$, with properties:

1. *Quasilinear* shuffling graph size. Given $n = \max(|S|, |T|)$, $|G| \in O(n \log n)$.
2. *Constant* maximum *fan-out* of each vertex, δ_{out}.
3. *Constant* maximum *fan-in* of each vertex, δ_{in}.

Table 1: HD Shuffle parameters

Constants
δ_{in}: maximum *fan-in* of each vertex
δ_{out}: maximum *fan-out* of each vertex
Arrays
$S = \{s_1, \ldots, s_p\}$: source partitions
$T = \{t_1, \ldots, t_q\}$: target partitions
Matrices
$T \to [\tau_1, \ldots, \tau_r]$: δ_{out}-factorization of $|T|$
$S \to [\sigma_1, \ldots, \sigma_d]$: δ_{in}-factorization of $|S|$

We present the Hyper Dimension Shuffle algorithm for shuffling data from source partitions $S = \{s_1, \ldots, s_p\}$ to target partitions $T = \{t_1, \ldots, t_q\}$. Then we prove the correctness of its properties.

A summary of this algorithm is: first factorize the input and output partitions into multi-dimensional arrays, and then execute multiple iterations where stages in each iteration aggregate data for one dimension and partition data for the next. Before outlining the full algorithm in section 3.5, we go over the operators that are used as building blocks. The parameters used in this section are listed in Table 1 for quick lookup.

3.1 Preliminaries

First we introduce our formal framework for partitioned data processing and describe the three operators that HD shuffle uses: hyper partitioner, dimension aggregator, and reshaper. Consider initial partitioned rowset $S = \{s_1, \ldots, s_p\}$ with p files. We define the *shape* of that rowset using an array of r dimensions. For example, if p is 8, we may use $r = 2$, and a dimension signature $[3, 3]$. This can be thought of as a file naming scheme for storing a set of up to $3 \times 3 = 9$ files, of which 1 slot is unused. While operators compute rowsets from rowsets, partitioned operators compute shaped rowsets from shaped rowsets. Hyper partitioners add one dimension, while dimension aggregators remove one. By convention, we add least-significant dimensions and remove most-significant.

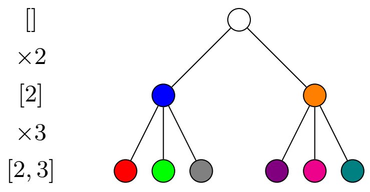


Figure 5: Hyper partition tree for $|T| = 6$ and $\delta_{out} = 3$

Figure 5 shows a degenerate case of an HD shuffle graph with only hyper partitioners. It is composed of two hyper partitioners partitioning a single zero-dimensional file into an intermediate result of two files of shape $[2]$, then into a final rowset with six files of shape $[2, 3]$. This HD shuffle graph can be uniquely identified by its input and output shape signatures, $[] \to [2, 3]$. This graph is full, but the algorithm extends to incomplete graphs with unused slots.

The dimension aggregator aggregates one dimension. The first aggregator in the HD shuffle graph in Figure 6 aggregates a dimension size of 3, so each operator instance reads up to three files and outputs one. The algorithm used may be order-preserving

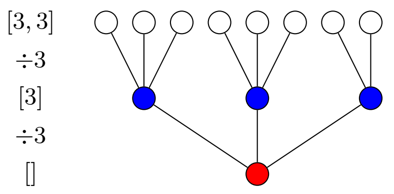


Figure 6: Dimension aggregation tree with $|S| = 8$ and $\delta_{in} = 3$

or non-order-preserving, depending on whether the entire shuffle needs to be order-preserving. This graph is the HD shuffle with signature $[3, 3] \to []$ for when input size $|S| = 8$. Reshaping $|S|$ into the 2-dimensional array $[3, 3]$ leaves one unused slot at the end.

The reshape operator coerces a rowset into a desired shape without moving or actually renaming any files. It purely establishes a mapping from one shape to another. Consider a shuffle into $|T| = 6$ partitions. Given a *fan-out* limit for the shuffle of $\delta_{out} = 3$, there are a few possible choices of r and specific dimension sizes that create a legal shuffle graph, but they all have $r > 1$, since $|T| > \delta_{out}$. Say we choose a target shape of $[2, 3]$. Depending on what happens after the shuffle, this two-dimensional shape may no longer be convenient, but instead a shape of $[6]$ may be desired for further processing. The reshape operator does exactly that. We may need to use it at the beginning and end of an HD shuffle.

3.2 Factorization and the Reshape Operator

The reshape operator maps one shape $[n_1, \dots, n_r]$ to another. Creating a shape from a cardinal number we call factorization.

Definition 1. An *α-factorization* of $[n]$ into $[n_1, \dots, n_r]$ must satisfy:

$$r \in O(\log n)$$
$$1 < n_i \le \alpha,\ \forall i \in \{1, \dots, r\}$$
$$n \le \prod_{i=1}^{r} n_i < \alpha \times n$$

A δ_{in}-factorization of $|S|$ is constructed to shape source rowset S so it is ready for HD shuffle. For example, in figure 6, we factorized $|S| = 8 \to [3, 3]$. The δ_{out}-factorization of $|T|$ is the shape of the shuffle result, such as $[2, 3]$ in figure 5. The HD shuffling graph is uniquely determined by these two shapes. The definition stipulates some bounds that both serve to keep the size of the graph and the *fan-out* and *fan-in* of the nodes bounded, at the cost of increased iterations. One simple way to construct a valid α-factorization is to select a minimal dimensionality r such that $\alpha^r \ge n$, and set all $n_i = \alpha$.

3.3 Hyper Partition Operator: Partitioning by Dimension

Let $[\tau_1, \dots, \tau_r]$ be the δ_{out}-factorization of $|T|$. This generates r iterations of the hyper partitioner. Iteration $i \in \{1, \dots, r\}$ increases the number of dimensions by one by appending a dimension of length τ_i. The operator computes the index of the intermediate file where a tuple belongs by computing the function:

$$P(k, i) = \left(\frac{k}{\prod_{j=i+1}^{r} \tau_j} \right) \bmod \tau_i$$

Each tuple starts with a known partition key $k \in \mathbb{N}_{|T|}$ indicating its target partition. At the beginning of shuffling, that is uncorrelated with its location. Then each iteration increases knowledge about where that tuple is located by one dimension. The file a tuple lands in corresponds to a single path from the root of the graph in figure 5, which also corresponds to the sequence of P values up to that point, culminating in the sequence $[P(k, 1), \dots, P(k, r)]$. Each color in the figure indicates a different path, and therefore a different state of information. Note that τ_i is the *fan-out*, and by definition $\tau_i \le \delta_{out}$. After r partition iterations, all the tuples in any given file share the same value of k. Since the definition of the δ_{out}-factorization of $|T|$ guarantees that there are at least $|T|$ different files after r iterations, that is sufficient to separate all values of k.

3.4 Dimension Aggregation Operator: Merging by Dimension

The tuples destined for a target partition $t_j \in T$ can start out scattered across all the source partitions $s_i \in S$. Let $[\sigma_1, \dots, \sigma_d]$ be the δ_{in}-factorization of $|S|$. This gives us d iterations for the dimension aggregator, which removes one dimension σ_i from the shape of the intermediate rowset in each iteration $i \in \{1, \dots, d\}$.

Figure 6 shows the aggregation process from $|S| = 8$ source partitions to a single target, with *fan-in* limit $\delta_{in} = 3$, and δ_{in}-factorization $[3, 3]$. Each $\sigma_i \le \delta_{in}$ in the factorization is used as the *fan-in* of a vertex.

Aggregation does not lose any knowledge about where tuples are located. That is because we only aggregate files with other files that have the same state of knowledge about the destinations of the tuples they contain.

3.5 Algorithm

Algorithm 1 Hyper Dimension shuffling

```
 1: function HD SHUFFLE($S, T$)
 2:     $[\tau_1, \dots, \tau_r] \leftarrow \delta_{out}$-factorize($|T|$)
 3:     $[\sigma_1, \dots, \sigma_d] \leftarrow \delta_{in}$-factorize($|S|$)
 4:     $B_0 \leftarrow$ reshape($S, [\sigma_1, \dots, \sigma_d]$)
 5:     $\{i, j, k\} \leftarrow \{1, 1, 1\}$
 6:     for $i \le r$ or $j \le d$ do
 7:         if $i \le r$ then
 8:             $B_k \leftarrow$ hyperpartition(
 9:                 $B_{k-1}$,
10:                 $[\sigma_j, \dots, \sigma_d, \tau_1, \dots, \tau_i]$)
11:             $i \leftarrow i + 1$
12:         else
13:             $B_k \leftarrow B_{k-1}$
14:         end if
15:         if $j \le d$ then
16:             $B_k \leftarrow$ dimensionaggregate(
17:                 $B_k$,
18:                 $[\sigma_{j+1}, \dots, \sigma_d, \tau_1, \dots, \tau_{i-1}]$)
19:             $j \leftarrow j + 1$
20:         end if
21:         $k \leftarrow k + 1$
22:     end for
23:     $T \leftarrow$ reshape($B_{k-1}, [|T|]$)
24: end function
```

The pseudocode of the HD shuffle algorithm, from source partitions S to target partitions T, is presented in algorithm 1.

HD shuffle algorithm begins with computing a δ_{out}-factorization of $|T|$ into $[\tau_1, \dots, \tau_r]$ at line 2, and a δ_{in}-factorization of $|S|$ into

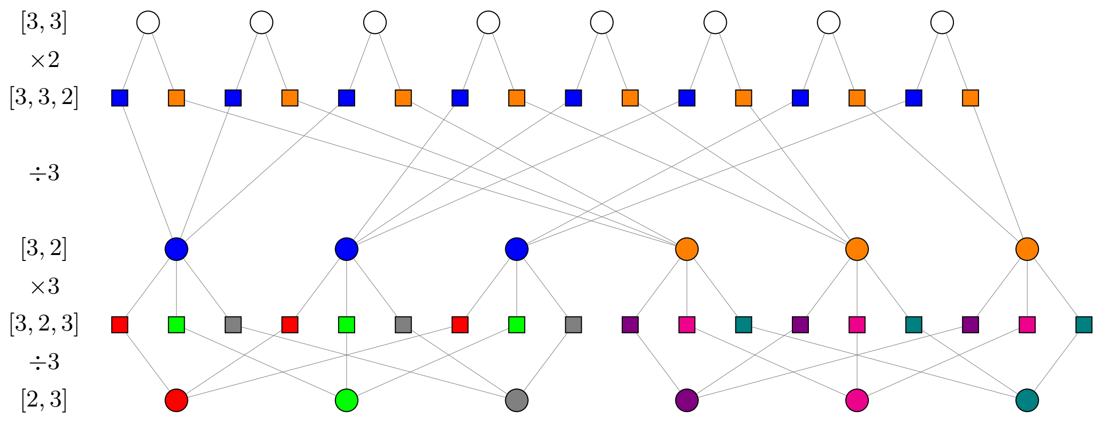


Figure 7: Hyper Dimension Shuffle from $|S| = 8$ to $|T| = 6$ with $\delta_{in} = 3$ and $\delta_{out} = 3$

$[\sigma_1, \ldots, \sigma_d]$ at line 3. To ensure that the size of the intermediate vertices doesn't exceed $O(\max(|S|, |T|))$ during any data shuffle iteration, the algorithm requires the additional constraint $\tau_i \leq \sigma_i$ for $i \in \{1, \ldots, \min(r, d) - 1\}$. It then utilizes the reshape operator to map the sources S into the d-dimensional array $[\sigma_1, \ldots, \sigma_d]$ represented by B_0, at line 4.

During iteration k, where $1 \leq k \leq r$, we partition the data of each vertex from the shaped rowset B_{k-1} into τ_k output files using the hyper partitioner, at line 8. The shaped rowset B_k is generated from the shape of B_{k-1} by adding a dimension of length τ_k.

During iteration k, where $1 \leq k \leq d$, the dimension aggregator collects σ_k files together into one, thereby removing one dimension of length σ_k from B_k at line 16.

Since a partitioner that follows an aggregator uses the same shape B_k of files as its input, we can combine the two operators to be executed within a single process for efficiency.

After $k-1 = \max(r, d)$ iterations, the HD shuffle algorithm has a resulting rowset B_{k-1} of shape $[\tau_1, \ldots, \tau_r]$. Some of the files can be empty, as the space can have holes depending on the δ_{out}-factorization of $|T|$. However, according to the hyper partitioner, the first $|T|$ files will represent the union of all data that we started in S. As such, we reshape B_{k-1} into the 1-dimensional array $[|T|]$ and return that as our result, at line 23.

THEOREM 1. *The Hyper Dimension shuffling algorithm, from sources S to targets T with constant maximum* fan-in δ_{in} *and* fan-out δ_{out}, *generates a graph of complexity $|G| = |V| + |E| \in O(n \log n)$, where $n = \max(|S|, |T|)$.*

PROOF. Let the δ_{out}-factorization of $|T|$ be $[\tau_1, \ldots, \tau_r]$, and the δ_{in}-factorization of $|S|$ be $[\sigma_1, \ldots, \sigma_d]$, with the additional constraint that $\tau_i \leq \sigma_i$ for $i \in \{1, \ldots, \min(r, d) - 1\}$.

Since $r \in O(\log |T|)$ and $d \in O(\log |S|)$ and the algorithm has $\max(r, d)$ iterations, then we have at most $O(\log n)$ iterations.

The number of vertices used during iteration k is upper bound by the number of elements within B_k. As this multi-dimensional array is reshaped at most twice during one iteration, we can consider the maximum size of B_k as the complexity of vertices for a given k.

During iteration $k \in \{1, \ldots, \min(r, d)\}$, the size of B_k is at most:

$$|S| \times \tau_k \times \prod_{i=1}^{k-1} \frac{\tau_i}{\sigma_i} \leq |S| \times \tau_k \leq \delta_{out} \times |S| \in O(n)$$

If $r > d$, then for iteration $k \in \{d+1, \ldots, r\}$ the size of B_k is at most:

$$|S| \times \prod_{i=1}^{d} \frac{1}{\sigma_i} \times \prod_{i=1}^{k} \tau_i \leq \prod_{i=1}^{k} \tau_i \leq \delta_{out} \times |T| \in O(n)$$

If $r < d$, then for iteration $k \in \{r+1, \ldots, d\}$ the size of B_k is at most:

$$|S| \times \prod_{i=r+1}^{k} \frac{1}{\sigma_i} \times \prod_{i=1}^{r} \frac{\tau_i}{\sigma_i} \leq |S| \times \prod_{i=r+1}^{k} \frac{1}{\sigma_i} \leq |S| \in O(n)$$

Hence, during any iteration of the algorithm, the number of used vertices is $O(n)$. The total number of vertices in the graph is the product of the number of iterations and the number of vertices used within an iteration. Hence $|V| \in O(n \log n)$.

Each vertex in the graph has a constant maximum *fan-in* of δ_{in} and *fan-out* of δ_{out}. Then, we have that:

$$|E| \leq (\delta_{in} + \delta_{out}) \times |V| \in O(n \log n)$$

Therefore, $|G| = |V| + |E| \in O(n \log n)$. $\square$

3.6 Example

We illustrate the HD shuffle algorithm through an example shown in figure 7. In this example we shuffle $[3, 3]$ to $[3, 2]$, and show how the partitioning tree in figure 5 interleaves with the aggregation tree in figure 6. We shuffle data from $|S| = 8$ source partitions to $|T| = 6$ target partitions, using a maximum *fan-in* and *fan-out* of value $\delta_{in} = \delta_{out} = 3$.

When we interleave partitioners and aggregators, we need not materialize after aggregation. Instead, the data is piped straight into the next partitioner. This diagram emphasizes this by representing a vertex as a circle ◯ whether it contains an aggregator, a partitioner, or both. A square ☐ represents an the intermediate file that does need to be sent to another vertex.

The algorithm computes the δ_{out}-factorization of $|T|$ as $[\tau_1, \tau_2] = [2, 3]$ and the δ_{in}-factorization of $|S|$ as $[\sigma_1, \sigma_2] = [3, 3]$. Every tuple has a partitioning key with value in $\mathbb{N}_6$ that indicates the final target partition it belongs to. We can view these key values in base 3, to align the actions taken by the hyper partitioner on dimensions with the values of the digits within the keys.

In the first iteration, each of the 8 source vertices partition their data into 2 intermediate files labeled by different colors indicating the value of P(k,1) of the tuples they contain. This creates a total of 16 physical files of shape $[\sigma_1, \sigma_2, \tau_1] = [3, 3, 2]$. Then, the dimension aggregator removes the σ_1 dimension by combining 3 of the files into one collection of tuples, generating new intermediate vertices of shape $[\sigma_2, \tau_1] = [3, 2]$. Each vertex aggregates a distinct set of files that all have the same set of final destinations.

The next shuffling iteration proceeds by having the same vertices that aggregated utilize the hyper partitioner to each produce 3 intermediate files, where each file corresponds to a fixed key value in $\mathbb{N}_6$. That results in 18 files with shape $[\sigma_2, \tau_1, \tau_2] = [3, 2, 3]$.

Finally, the dimension aggregator uses a 2-dimensional shape $[\tau_1, \tau_2] = [2, 3]$ to remove the σ_2 dimension, by combining the 3 files for each unique partitioning key into a single file. Thus, all the tuples that share the same target partitioning key value are together in their own target file.

4. HD SHUFFLE IN SCOPE

HD shuffle is in production in the SCOPE big data analytics engine both for batch and streaming jobs. SCOPE's Job Manager (JM) inherits from Dryad [21] the dimensional model of partitioned processing. Edges in the data flow graph are connected according to mappings between the upstream and downstream vertices. We implement the reshape operator as such a logical mapping. Dimension aggregator is a variation on SCOPE's merger that reads data from upstream vertices in the aggregating dimension. The hyper partitioner is a SCOPE partitioner extended to support the recursive hyper partitioning function.

4.1 SCOPE Execution Model

SCOPE is part of the Cosmos big data storage and analytics platform. It is used to analyze massive datasets by integrating techniques from paralleled databases [11] and MapReduce systems [7, 17] with a SQL-like declarative query language called SCOPE. A SCOPE script is compiled into a query execution plan using a cost-based optimizer derived from Cascades [20]. A query execution plan is a DAG of parallel relational and user-defined operators. Graph fragments are grouped into pipelines, and a DAG of data flow stages is superimposed on the query execution plan. This stage graph is then expanded by JM into the full graph of vertices and scheduled on the available resources in Cosmos clusters. In batch mode, channels are blocking edges that write a file to local storage, ready for downstream vertices to read remotely. Exploiting the pull execution model and blocking to write to local SSD costs time, but gives batch jobs high resiliency and fault tolerance, since compute resources can be acquired whenever they become available, while failed vertices can be rerun.

Figure 8a shows a stage graph of a simple query which reads data from unstructured inputs and outputs into a structured stream with a specific partitioning property. There are two stages, `SV1` and `SV2`, with two operators in each. The query is first translated into a logical tree containing two operators: the extractor and the outputter. The optimizer generates an execution plan with a additional partitioner and merger, physical operators that together implement shuffle, enforcing the partitioning required downstream. The partitioning S of the input data cannot satisfy the required partitioning T of the structured stream. The degree of parallelism (abbreviated as DOP), the number of independent executions (vertices) of each stage is determined by the optimizer. In this case the DOP of the first stage could be a function of the data size of the input, and the DOP of the second stage could be inherited from the structured stream's required partitioning. In the naïve shuffling model from Figure 1, each downstream merger depends on every upstream partitioner. Scaling out the data processing leads to high *fan-out* and *fan-in*. The HD shuffle implementation as in figure 8b trades off

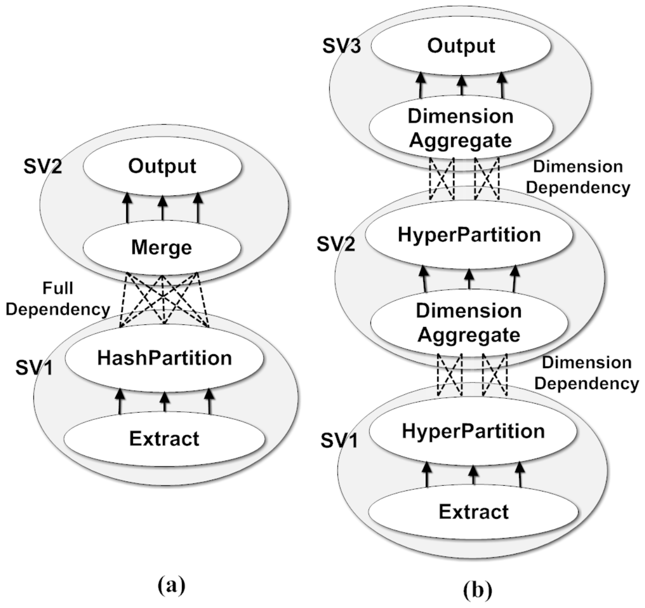


Figure 8: SCOPE execution graphs

path-length for fixed *fan-out* and *fan-in*. Another iteration of partition and merge can be added to scale out further.

The HD shuffle algorithm is used to replace the existing index-based partitioning and dynamic aggregation techniques [13] which are originally proposed to address high *fan-out* and *fan-in* in SCOPE. Index-based partitioning is triggered when the output partitions exceeds the system *fan-out* limit while dynamic aggregation is triggered when the input partitions exceeds the system *fan-in* limit. Using the same criteria, HD shuffle is triggered when the system *fan-out* and *fan-in* limits are exceeded by the data shuffle operation. In SCOPE, the best choice of *fan-out* and *fan-in* limits are determined by several factors. Both reads and writes reserve I/O buffer memory for each stream. By default, the unit of Cosmos computing resource (called a token) has `6GB DDR4-2400` heap memory, each input stream reserves `16MB`, and each output stream reserves `4MB`. This creates an upper bound of *fan-in* < 375 and *fan-out* < 1500 for a one-token container. Another factor is the multiplication of the probability of connection failures observed in Cosmos clusters. In practice, when the cluster is experiencing load and maintenance outages at the high end of normal range, jobs with *fan-in* in the thousands stop making progress. Currently, we use 250 as the default *fan-in* limit and 500 as the default *fan-out* limit for HD Shuffle in SCOPE.

HD shuffle can increase path length because r, the number of iterations, may increase as needed, and each iteration adds a layer of tasks rearranging and materializing the same data. However, HD shuffle only requires two iterations of partition and aggregation stages on SCOPE's current workload, given the system *fan-out* and *fan-in* limits, which is no more than the current dynamic aggregation approach. So for SCOPE, HD Shuffle is a pure win. In other systems, some shuffle algorithms require no extra data materialization by utilizing a sliding aggregating window for the input channels shown in figure 3. For those systems, there is a trade-off between extra data materialization of HD shuffle and the overhead of excessive remote fetches and data spilling of the intermediate aggregating results. We demonstrate this overhead is dominating in Spark and HD shuffle improves the shuffle performance through our experiments.

4.2 SCOPE Optimizations for HD Shuffle

HD shuffle takes advantage of several optimizations in SCOPE to improve scheduling efficiency and query plan quality.

4.2.1 Job Scheduling Efficiency

When the inputs and outputs of a shuffle are not whole multiples of *fan-in* and *fan-out*, the α-factorization generates a sparse array with unused slots. We leverage *dummy vertex optimization* to support sparse array scheduling. During job graph construction, some mergers will have missing inputs, which we represent as dummy channels. During execution, unnecessary connections are ignored and unnecessary vertices are skipped.

Another optimization that we use is *group-aware scheduling*. For the naïve shuffling model, the downstream merger cannot be executed until all upstream partitioners complete. This introduces long tail effects on job execution where all downstream vertices are stuck waiting for a few upstream stragglers. HD shuffle has a key advantage: the downstream merger depends only on the partitioners with the same value in one dimension. Thus the barrier between successive stages is scoped down to that dimension, and waves can be scheduled independently.

4.2.2 Removing Unnecessary Stage Breaks

HD shuffle as described introduces a reshape operator at the beginning and end. The reshape operator is just a renumbering of channels that does not need to move any data. We observe that the reshape operator introduces an extra vertex boundary in some jobs incurring additional data materialization. Materialization just for bookkeeping purposes would be unfortunate. In the example shown in figure 8b, the reshape operators would occur in the middle of stages `SV1` and `SV3`, breaking them in half. We solve this by reordering reshape operators with the other operators in the execution graph. In the example, the reshape operators are moved after the first hyper partitioner and before the last dimension aggregator, avoiding any additional stage breaks.

4.2.3 Repetition of Data Reducing Operations

It is efficient to reduce data early before data movement, so SCOPE has built support for local GroupByAggregate, user-defined recursive reducer, and TopNPerGroup. Since HD shuffle moves data multiple times, there is an opportunity to apply the operation each time. For example, if a sum is desired, once one round of aggregation is done, some rows in the same group may have moved to the same vertex, so a round of local sums there can reduce data further before sending the rows to their destination (where another sum will be done).

We have not found enough evidence that using such mechanism would make an appreciable improvement so we have not enabled it yet for HD shuffle.

4.3 StreamSCOPE Compatibility

The naïve data shuffle model prevents scaling to the needs of SCOPE's largest streaming jobs. Two key requirements of streaming applications are continuous data processing and fast recovery from faults and stragglers [27]. For continuous data processing, SCOPE implements a variant with streaming channels. For fast recovery, SCOPE checkpoints all channels in globally addressable highly-available storage. The memory requirements of sorted merger with streaming input channels imposes a hard constraint on *fan-in*. Meanwhile, checkpoints built on the quadratic partitioning intermediate files causes excessive small writes.

The HD shuffle algorithm is a key enabler for the production SCOPE streaming pipelines. The *fan-in* limit, which guarantees the maximum number of upstream vertices for dimension merger, allows tight system control over the memory used by streaming channels. The number of global files written by checkpoints of the hyper partitioners can be tuned by choosing the proper partition dimension constrained by the *fan-out* limit. The flexibility to choose different dimension lengths of the α-*decomposition* allows for different heuristics to balance the number of intermediate vertices and the workloads. When the number of recursive partition and aggregation iterations is held constant, the number of vertices in the streaming job is linear in the number of input and output partitions. Compared to the naïve shuffle, HD shuffle constrains the dependency between a node and its inputs to a single dimension, so it is less expensive to recover from faults and stragglers. SCOPE is now able to support streaming jobs with $|S| = |T| = 2000$.

5. EVALUATION

We have implemented the Hyper Dimension Shuffle algorithm in the SCOPE [13] language (referred to as SCOPE+HD in this section), and deployed it into the production Cosmos platform used inside Microsoft. The evaluation is performed for both synthetic benchmarks and production workloads.

The unit of computing resource allocated in Cosmos clusters is a token, representing a container composed of $2\times$ `Xeon E5-2673 v4 2.3GHz` vCPU and `6GB DDR4-2400` memory, on a machine with a `40Gb` network interface card. A task must start by acquiring a token and hold that token through its completion. The job parallelism, the number of concurrent tasks, is constrained by the number of tokens allocated to the job.

5.1 Synthetic Benchmark Evaluation

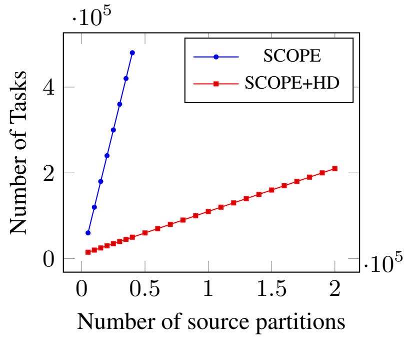
We use the TPC-H [6] benchmark for the synthetic evaluation and perform the evaluation for two different workloads: data shuffle jobs and TPC-H benchmark queries.

5.1.1 Scaling Input and Output Partitions

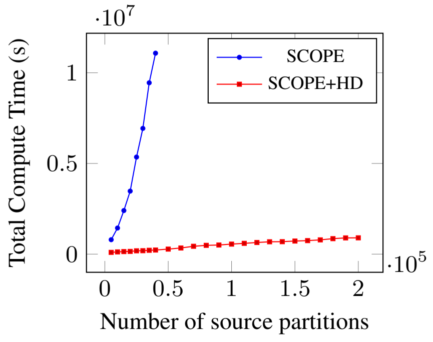
For this experiment, data is generated for TPC-H scale 10^3 (1 TB), with the `LINEITEM` table partitioned by the `L_ORDERKEY` column. To illustrate a large shuffling graph on a small payload, we read the `L_PARTKEY` column from the input table and project away the other columns. We repartition the data on `L_PARTKEY` column to produce the output table. The maximum *fan-out* and *fan-in* limits are set as $\delta_{out} = \delta_{in} = 500$. Each job uses 500 tokens. We perform two tests to scale the input and output partitions separately to better understand the performance impacts. For scaling the input partitions, we vary the number of source partitions $|S|$ from 5×10^3 to 200×10^3 incrementally by 5×10^3, while maintaining a constant number of target partitions $|T| = 5 \times 10^3$, and vice versa for scaling the output partitions. Three important metrics are recorded: 1. the total number of tasks performed in the job; 2. the end-to-end job latency; 3. the total compute time calculated by accumulating inclusive CPU utilization time reported by all SCOPE operators.

Figures 9a, 9c and 9b show what happens as the number of input partitions increases. While SCOPE does not reasonably scale past $|S| = 4 \times 10^4$ partitions, we observe that SCOPE+HD can easily go up to $|S| = 2 \times 10^5$ partitions. The number of tasks in both algorithms increases linearly with the number of input partitions, but SCOPE's slope is $|T|/250$ and SCOPE+HD's slope is 1. In SCOPE+HD, the number of tasks in the first round of partitioning is $|S|$, while the number of tasks in each subesequent round is fixed.

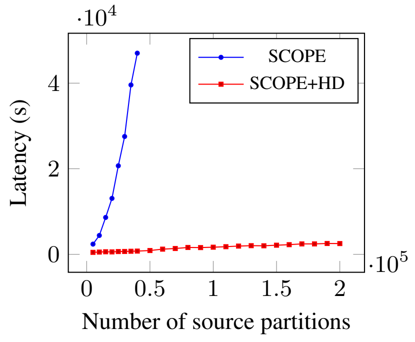
While the number of tasks directly affects the end-to-end job latency and the total compute time, it is not the only factor. For example, when $|S| = 1.5 \times 10^4$, SCOPE uses $\approx 1.8 \times 10^5$ tasks for the entire job, and has a job latency of $\approx 8.6 \times 10^3$ seconds. In



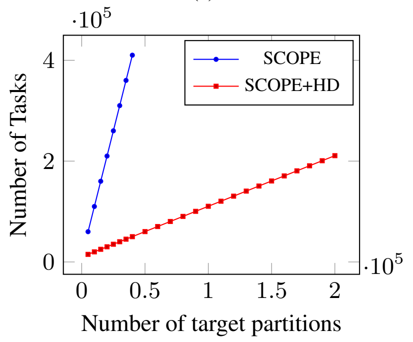
(a)



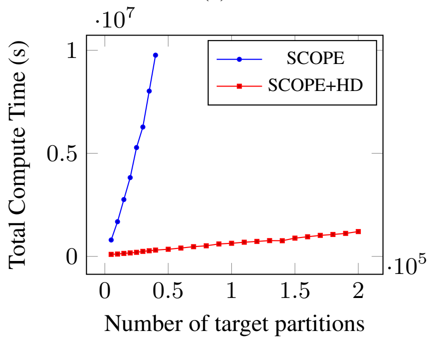
(b)



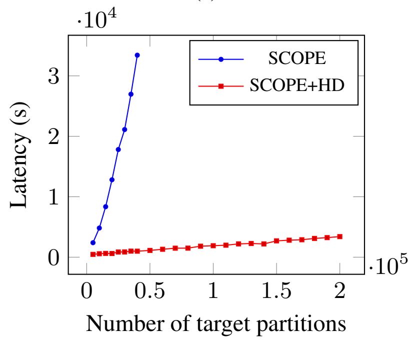
(c)



(d)

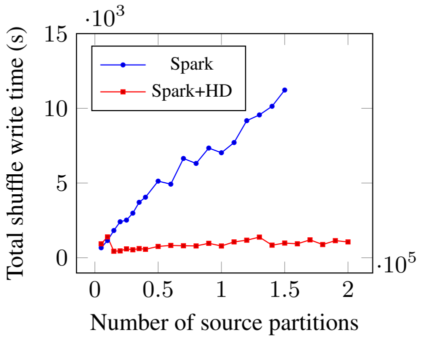


(e)

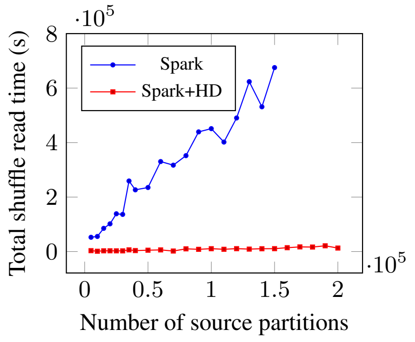


(f)

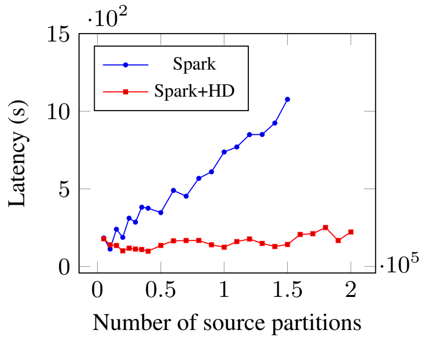
Figure 9: TPC-H scale 10^3, $\delta_{in} = \delta_{out} = 500$



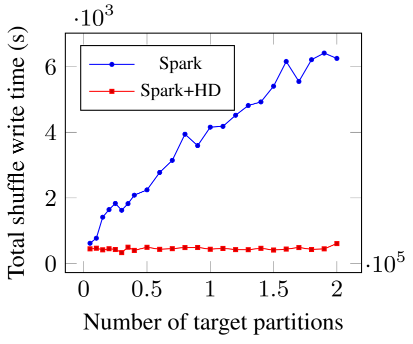
(a)



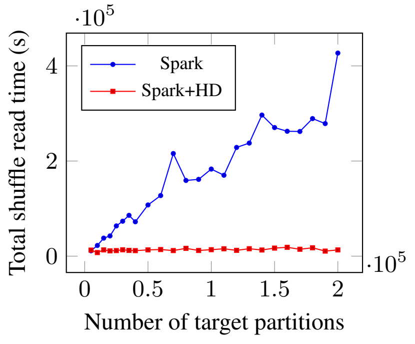
(b)



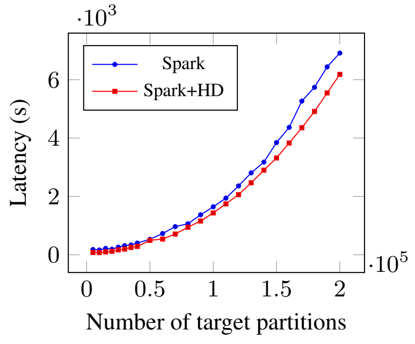
(c)



(d)



(e)



(f)

Figure 10: TPC-H scale 10^3, $\delta_{in} = \delta_{out} = 500$

contrast, when $|S| = 2 \times 10^5$, SCOPE+HD uses a larger number of tasks of $\approx 2.1 \times 10^5$, but shows a smaller job latency of $\approx 2.5 \times 10^3$ seconds. This is achieved by 1). using faster operators without the generation of a sorted indexed intermediate file, which translates in faster tasks; and 2). not fully partitioning a source in one iteration, which yields a higher data payload to connection ratio.

Figures 9d, 9f and 9e show that varying the number of target partitions $|T|$ results in similar behavior to what we see when we vary the number of source partitions.

We also look at the effect of scaling partition numbers on systems that limit the simultaneous count of opened connections, and only use one task per target partition to merge shuffled results, such as in figure 3. To that end, we implement HD shuffle in Spark 2.3.0 and use an HDFS shim layer on top of the Cosmos storage system. A session consisting of 500 executors is created with each executor matching a token. We run the same experiments as the ones described in Section 5.1.1 with baseline Spark and Spark+HD (we use $\delta_{out} = \delta_{in} = 500$ and $r = 2$ for Spark+HD). We compare the shuffling performance between Spark and Spark+HD and record three important metrics: total shuffle write time, total shuffle read time, and end-to-end job latency.

Figures 10a, 10b and 10c show the three metrics of scaling input partitions. We experience high job failure rate for baseline Spark when the job scales past $|S| = 16 \times 10^4$ input partitions. As the downstream merger requires data from all upstream partitioners, the multiplication of the probability of remote fetch failures due to unavailability of upstreams, frequently as a result of local file missing and network connection failures, becomes non-negligible. With the increase of upstream partitioners, the job will eventually fail due to too many retries for the downstream merger. In contrast, the dimension merger in HD shuffle only requires data from at most δ_{in} upstream tasks and it can scale up to $|S| = 2 \times 10^5$ partitions without any job failures observed. In terms of performance, Spark+HD achieves an almost constant running time reported by all three metrics, but baseline Spark has a linear increase of running time with the increase of input partitions. HD shuffle improves both reliability and performance for Spark.

Figures 10d, 10e and 10f show the effects of varying the number of target partitions $|T|$, while maintaining a constant number of source partitions. The same trend is found for the shuffle write and read metrics. However, the end-to-end job latency increases for both baseline Spark and Spark+HD. The job latency of Spark+HD improves by only about 15% compared with baseline Spark but shows a similar slope. This can be explained by a new bottleneck other than shuffling: the time spent creating the partitioned Parquet files in Cosmos global store. The Cosmos storage layer is designed with metadata operations that operate on a set of files. While SCOPE is able to leverage this implementation, Spark uses the metadata operations native to HDFS, through the HDFS shim layer. We observe that the per file metadata operation cost [2], in the Cosmos store, associated with the finalization of the increasingly partitioned output parquet file dominates job latency, when compared to the constant cost paid for the fixed partitioned count output parquet file in figure 10c.

5.1.2 *Scaling Shuffled Data*

Next, we study how the data shuffle algorithm scales when the amount of data to be shuffled increases while keeping per partition data size constant. We run jobs that use maximum *fan-out* and *fan-in* limits of $\delta_{out} = \delta_{in} = 250$ and `20GB` of data per input and output partition. The input data, the working set for the jobs, is the unordered data generated by the `DBGEN` tool for the `LINEITEM` table. This working set is then shuffled, to be partitioned by the `L_ORDERKEY` column. The data scalability tests are performed in two ranges since the original shuffle algorithm of SCOPE doesn't scale well beyond 100 TB: 1. SCOPE vs. SCOPE+HD: TPC-H scales ranging from 20×10^3 (20 TB) to 100×10^3 (100 TB) using 500 tokens; 2. SCOPE+HD only: TPC-H scales ranging from 20×10^4 (200 TB) to 100×10^4 (1 PB) using 2000 tokens.

For the first scalability test (20 TB $\sim$ 100 TB), SCOPE has a number of intermediate aggregator tasks quadratic in the TPC-H scale, as shown in figure 11a. Meanwhile, SCOPE+HD stays at a linear pattern with one recursive partitioning and aggregating iteration. However, the costs of scheduling and running intermediate aggregators which read and write tiny amount of data is overshadowed by those of the upstream partitioners and downstream mergers, as the task execution cost is dominated by the amount of data that a task processes. Thus, SCOPE is able to achieve a linear pattern in both latency and total compute time when the volume of data processed is correlated with the target partitioning count, as observed in figures 11c and 11b. SCOPE+HD, however, still yields a net benefit over SCOPE across these metrics, through avoiding the sort in SCOPE's index partitioner, reduced task scheduling, and reduced network connection overhead.

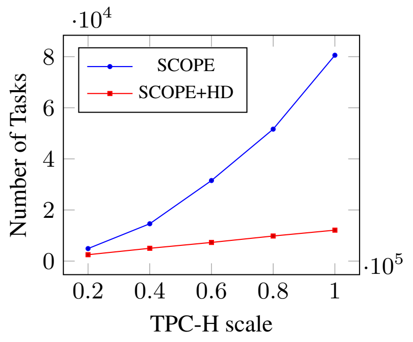
Finally, we examine the execution complexity for HD shuffle algorithm when the amount of shuffled data reaches petabyte levels. Figures 12a, 12c and 12b show the results for the SCOPE+HD jobs which shuffle the `LINEITEM` table in TPC-H scaling from 200 TB to 1 PB – the quadratic nature of SCOPE prohibits it from scaling gracefully when the number of partitions grows by an order of magnitude. We observe a linear pattern for the SCOPE+HD jobs, with the cost dominated by data transfer. It is also worth mentioning that no extreme stragglers of any task or increase of task failures have been observed in multiple runs of the 1 PB scale data cooking jobs. By providing tight guarantees on *fan-in* and *fan-out*, HD shuffle algorithm has proved to scale linearly in the number of tasks and computing resources, with room to go well beyond the 1 petabyte level.

5.1.3 *TPC-H Benchmark Queries*

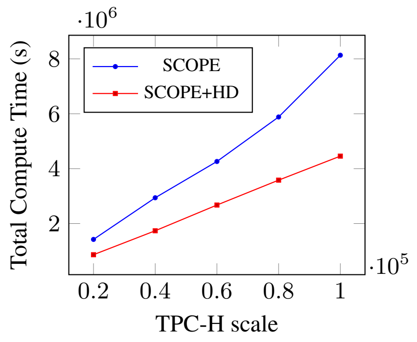
We run all 22 TPC-H benchmark queries on 100 TB TPC-H dataset using 1000 tokens to prove performance gains resulting from HD shuffle algorithm. The raw TPC-H data are first cooked into structured streams. Here, we use the following partitioning specification for the TPC-H dataset:

- The partitioning of the `LINEITEM` and `ORDERS` tables are aligned. The data are hash distributed by `ORDERKEY` values into 4000 partitions.
- The partitioning of the `PARTSUPP` and `PART` tables are aligned. The data are hash distributed by `PARTKEY` values into 1000 partitions.
- The `CUSTOMER` table is hash distributed by `CUSTKEY` into 500 partitions, and the `SUPPLIER` table is hash distributed by `SUPPKEY` into 100 partitions.

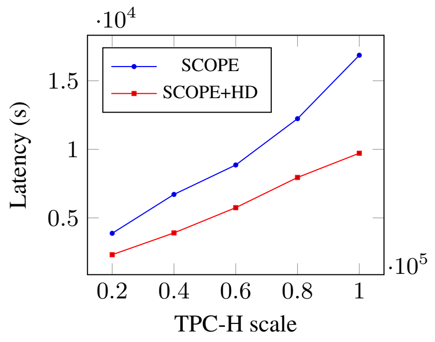
There are 8 out of 22 queries either requiring no data shuffle operation or requiring data shuffle operation with the input and output partitions below the set maximum *fan-out* and *fan-in* limits of $\delta_{out} = \delta_{in} = 250$. These queries are excluded from the comparison results as they don't trigger HD shuffle algorithm. The query performance results are shown in figure 13. Overall, HD shuffle algorithm achieves on average 29% improvement on the end-to-end job latency (figure 13b) and 35% improvement on the total compute time (figure 13a) for the 14 TPC-H queries. Particularly, HD shuffle algorithm achieves 49% improvement on



(a)

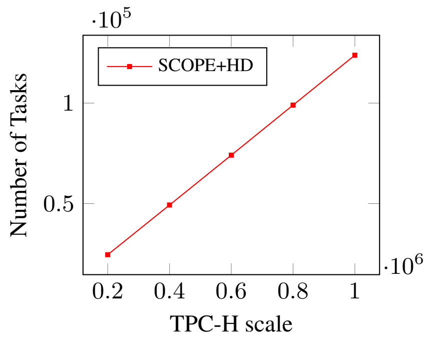


(b)

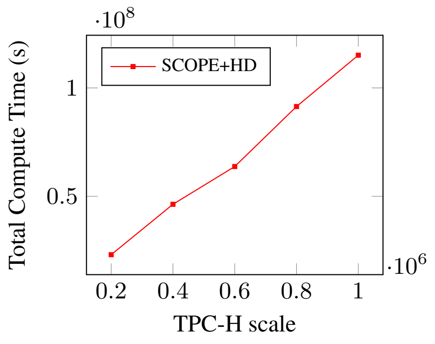


(c)

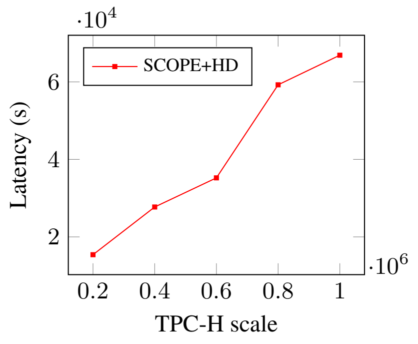
Figure 11: `20GB` per partition, $\delta_{in} = \delta_{out} = 250$



(a)

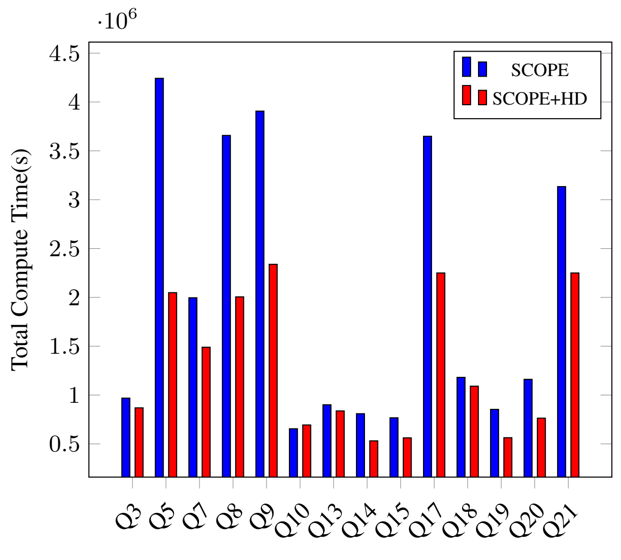


(b)

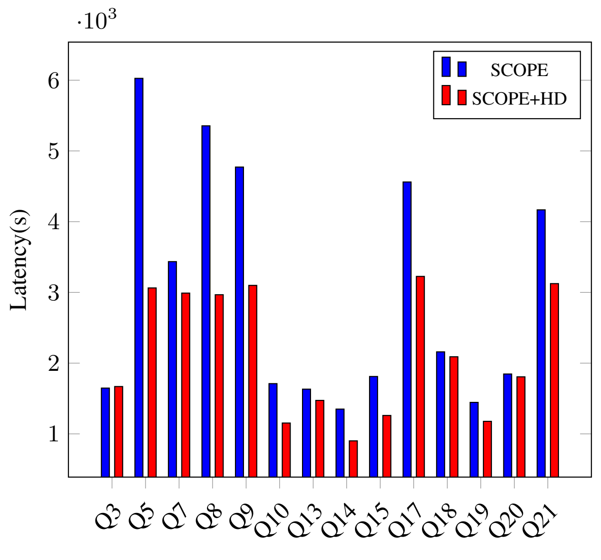


(c)

Figure 12: `20GB` per partition, $\delta_{in} = \delta_{out} = 250$



(a)



(b)

Figure 13: TPC-H scale 10^5

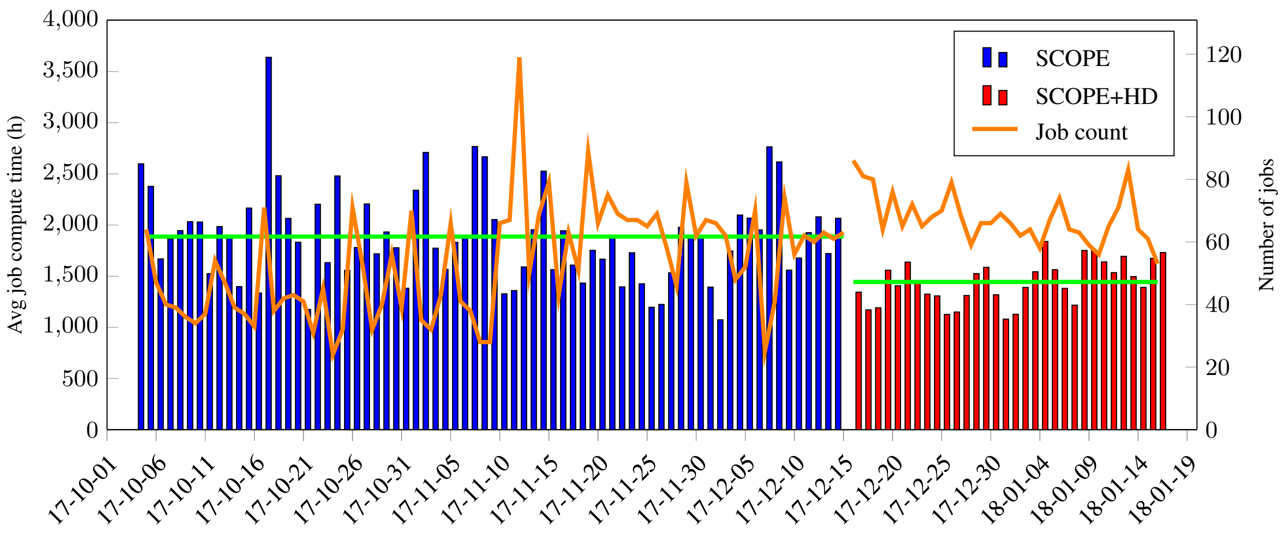


Figure 14: Improvement validation for recurring jobs

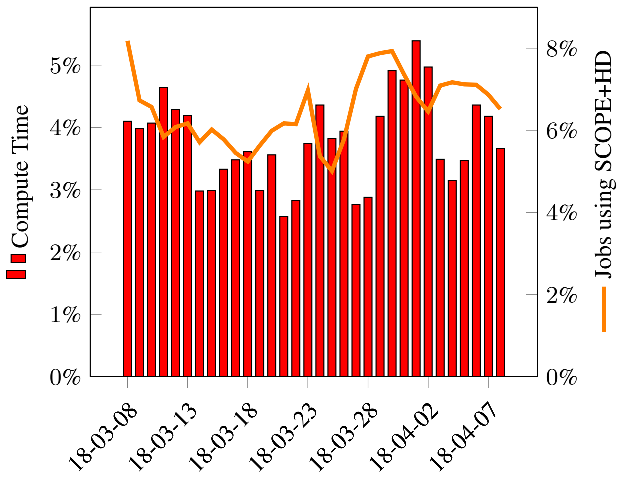


Figure 15: Cluster savings

the end-to-end job latency and 53% improvement on the total compute time for Q5, which is the largest performance gain among all queries.

We also observe that the end-to-end job latency and the total compute time are not always linearly dependent. For example, the total compute time of Q20 improves by 35% with HD shuffle algorithm but the end-to-end job latency only improves by 3%. It is because the data shuffle operation is not in the longest running path in the query execution graph which determines the job latency. Thus, the improvement from HD shuffle algorithm doesn't reflect on the the job latency for Q20. Another example is Q10 where the end-to-end job latency improves by 32% using HD shuffle algorithm but its total compute time degrades by 4%. The compute time degradation is a result of materializing more intermediate data. HD shuffle algorithm materializes all input data in the recursive partition and aggregate stages. In comparison, the dynamic aggregators are only scheduled for the input partitions over the *fan-in* limit and therefore materialize less data. For data shuffle operation with the input partitions slightly over the *fan-in* limit, the benefit of less data materialization overshadows the cost of extra sort operation for SCOPE without HD shuffle. However, the end-to-end job latency improves significantly due to the usage of non-blocking operator in HD shuffle algorithm which yields high scheduling efficiency. HD shuffle algorithm relieves the long tail effect caused by the stragglers of the partitioners which in turn improves the job latency.

5.2 Production Evaluation

Beyond the benchmark evaluation, HD shuffle algorithm also proves its success on real production workloads in SCOPE on Cosmos clusters, the internal big data analytics system at Microsoft. According to the SCOPE utilization statistics, data shuffling is the third frequently used operation and the most expensive physical operator in production workloads. The original data shuffle model in SCOPE utilizes the index-based partitioning and dynamic aggregation techniques. The former technique requires the partitioned data to be sorted in order to generated the indexed file and the latter introduces the quadratic number of intermediate aggregators. Both lead to inefficient utilization of compute resources and limits the amount of data which can be shuffled. To address these limitations, HD shuffle algorithm is implemented in SCOPE to provide an approximately linear performance guarantee when the shuffled data size increases. Here, we demonstrate the HD shuffle algorithm performance on real workloads through two evaluations [1]: 1. improvement validation for preview period; 2. efficiency improvement summary of HD shuffle feature after its general availability on all production clusters.

The improvement evaluation was performed from 12/16/2017 to 01/16/2018 on one of the production clusters for Microsoft Office team. By utilizing the workload analysis tool [23], we find the HD shuffle feature on that cluster enhances the performances of 26 recurring pipelines. Recurring pipeline schedules the same job periodically to process data from different time frames. All these candidate jobs contain one or more large data shuffling operations where the input and output partitions are beyond the system *fan-in* and *fan-out* limits. The performance evaluation result shown in figure 14 provides the performance comparison in terms of

[1]The production evaluation is not performed through A/B testing due to resource constraints, and the improvement is estimated based on the recurring job history.

the average compute hours (one compute hour means executing with one token for one hour) for the jobs from these 26 recurring pipelines before and after HD shuffle feature is enabled. As we highlighted in the figure, by enabling the HD shuffle feature the average compute hours for the jobs from these recurring pipelines reduce over 20% and there are around 60 jobs scheduled everyday which yields 3% token savings for that cluster.

SCOPE announced general availability of HD shuffle feature in March 2018. For the efficiency summary, we collect all jobs which utilized HD shuffle feature from 03/08/2018 to 04/07/2018 and compare the total compute hours of these jobs with the 75 percentile of their historic runs. The accumulated compute hour savings from these jobs are shown in figure 15. HD shuffle feature contributes to 4% cluster-wide savings by improving 7% of jobs which are enormous improvements for these jobs. For example, one of the shuffling-intensive production job has its total compute hours reduced from 8129 hours to 3578 hours (-56%) and its end-to-end job latency reduced from 5 hours to 3 hours. It is not surprising at all that HD shuffle is one of the largest efficiency improvement features which frees up over thousands of machines everyday.

6. RELATED WORK

In the background section, we briefly describe some techniques widely embraced by state of the art distributed systems [1, 13, 26, 29] to improve data shuffle operations. A comprehensive description of recently proposed techniques and optimizations follows.

SCOPE [13, 29] proposes two techniques to improve the scalability of data shuffle operations: index-based partitioning and dynamic aggregation. Index-based partitioning is a runtime operation which writes all partitioned outputs in a single indexed file. It converts the enormous random I/O operations to sequential writing and reduces the excessive number of intermediate files. On the other hand, the partition task requires a stable sort operation on the partition key column to generate the index file, which is an added expense and blocking operation. Dynamic aggregation is used to address the *fan-in* issue for mergers when the number of source inputs is large. The job scheduler [12] uses several heuristics to add intermediate aggregators at runtime dynamically, from which the downstream merger operator would be scheduled subsequently to combine the partial merged results to accrue the final ones. It guarantees that the amount of required memory buffer and concurrent network connections of any merger can be constrained at a desired level. It can also achieve load balancing, data locality and fast recovery for the intermediate aggregators. Despite these benefits, dynamic aggregation introduces quadratic intermediate aggregators, which causes significant scheduling overhead.

Hadoop and its relatives [1, 7, 18] have used similar techniques to write a singled indexed file and perform intermediate aggregations in the downstream mergers. Aligned with the purpose of this paper, we would like to highlight some optimizations on top of Hadoop which improve its data shuffle performance. Sailfish [25] is built around the principle of aggregating intermediate data and mitigating data skew on Hadoop by choosing the number of reduce tasks and intermediate data partitioning dynamically at runtime. It enables network-wide data aggregation using an abstraction called I-files which are extensions of KFS [5] and HDFS [4] to facilitate batch transmission from mappers to reducers. By using I-files and enabling auto-tuning functionality, Sailfish improves the shuffle operation in Hadoop. However, Sailfish is sensitive to stragglers of the long running reduce tasks and fragile to merger failures due to high revocation cost. Hadoop-A [22] improves Hadoop shuffle by utilizing RDMA (Remote Direct Memory Access) over InfiniBand. It addresses the two critical issues in Hadoop shuffle: (1) the serialization barrier between shuffle/merge and reduce phases and (2) the repeated merges and disk access. To eliminate the serialization overhead, a full pipeline is constructed to overlap the shuffle, merge and reduce phases for ReduceTasks, along with an alternative protocol to facilitate data movement via RDMA. To avoid repeated merges and disk access, a network-levitated merge algorithm is designed to merge data. However, the reduce tasks buffer intermediate data in memory, which enforces a hard scalability limit and incurs high recovery cost for merging failures. Optimizing the shuffling operations for distributed systems built with emerging hardware, such as NUMA machines is also explored in recent research [24]. The evidence shows that designing hardware-aware shuffle algorithms provides great opportunities to improve performance for distributed systems.

In Spark [26] and other related systems [8, 9, 14, 19] that rely on efficient in-memory operators, the cost of data shuffle using slow disk I/O operations is magnified. The proposed optimizations [16, 28] focus on providing efficient partial aggregation of the intermediate data for the map tasks. [16] has introduced shuffle file consolidation, an additional merge phase to write fewer, larger files from the map tasks to mitigate the excessive disk I/O overhead. Shuffle file consolidation refers to maintaining a single shuffle file for each partition, which is the same as the number of Reduce tasks, per core rather than per map task. In other words, all Map tasks running on the same core write to the same set of intermediate files. This significantly reduces the number of intermediate files created by Spark shuffle resulting in better shuffle performance. Despite the simple and effective idea, it remains unclear about the failure recovery mechanism and the way to handle stragglers of map tasks. Riffle [28] on the other hand proposes an optimized shuffle service to provide machine level aggregation, which efficiently merge partitioned files from multiple map tasks scheduled on the same physical machine. The optimized shuffle service significantly improves I/O efficiency by merging fragmented intermediate shuffle files into larger block files, and thus converts small, random disk I/O requests into large, sequential ones. It also develops a best-effort merge heuristic to mitigate the delay penalty caused by map stragglers. Riffle keeps both the original, unmerged files as well as the merged files on disks for fast recovery from merging failures. However, the merging operations need to provide read buffer for all intermediate files which becomes a system overhead. Rescheduling failed or slow map tasks on different machines will require merging the intermediate files from scratch. Slow map stragglers still have a long tail effect on job latency.

7. CONCLUSION

Motivated by the scalability challenges of data shuffle operations in SCOPE, we present an innovative data shuffle algorithm called Hyper Dimension Shuffle. By exploiting a recursive partitioner with dimension merger, HD shuffle yields *quasilinear* complexity of the shuffling graph with tight guarantees on *fan-in* and *fan-out*. It improves the current data shuffle model in SCOPE by using faster operators and avoiding the quadratically increasing shuffling graph. Based on our experiments, HD shuffle stands out among the current data shuffle algorithms through its competitive advantage in handling data shuffle operations at petabyte scale with remarkable efficiency for both synthetic benchmarks and practical application workloads.

8. REFERENCES

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

[2] Apache Spark. https://spark.apache.org/.
[3] Apache Tez. https://tez.apache.org/.
[4] Hadoop distributed filesystem. http://hadoop.apache.org/hdfs.
[5] Kosmos distributed filesystem. http://kosmosfs.sourceforge.net/.
[6] TPC-H specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.
[7] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.
[8] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: Interactive queries on very large data. *PVLDB*, 5(12):1902–1905, 2012.
[9] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.
[10] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling spark in the real world: Performance and usability. *PVLDB*, 8(12):1840–1843, 2015.
[11] J. A. Blakeley, P. A. Dyke, C. Galindo-Legaria, N. James, C. Kleinerman, M. Peebles, R. Tkachuk, and V. Washington. Microsoft sql server parallel data warehouse: Architecture overview. In M. Castellanos, U. Dayal, and W. Lehner, editors, *Enabling Real-Time Business Intelligence*, pages 53–64, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
[12] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 285–300, Berkeley, CA, USA, 2014. USENIX Association.
[13] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
[14] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. Quill: Efficient, transferable, and rich analytics at scale. *PVLDB*, 9(14):1623–1634, 2016.
[15] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.
[16] A. Davidson and O. Andrew. Optimizing shuffle performance in spark. In *Tech. Rep*, University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, 2013.
[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
[18] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.
[19] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.
[20] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.
[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.
[22] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
[23] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 191–203, New York, NY, USA, 2018. ACM.
[24] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.
[25] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.
[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.
[28] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.
[29] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, Oct. 2012.