# On Distributing Symmetric Streaming Computations

Jon Feldman* S. Muthukrishnan† Anastasios Sidiropoulos‡ Cliff Stein§
Zoya Svitkina¶

**Abstract**

A common approach for dealing with large data sets is to stream over the input in one pass, and perform computations using sublinear resources. For truly massive data sets, however, even making a single pass over the data is prohibitive. Therefore, streaming computations must be distributed over many machines. In practice, obtaining significant speedups using distributed computation has numerous challenges including synchronization, load balancing, overcoming processor failures, and data distribution. Successful systems in practice such as Google's MapReduce and Apache's Hadoop address these problems by only allowing a *certain class* of highly distributable tasks defined by local computations that can be applied in any order to the input.

The fundamental question that arises is: How does the class of computational tasks supported by these systems differ from the class for which streaming solutions exist?

We introduce a simple algorithmic model for massive, unordered, distributed (mud) computation, as implemented by these systems. We show that *in principle*, mud algorithms are equivalent in power to symmetric streaming algorithms. More precisely, we show that any symmetric (order-invariant) function that can be computed by a streaming algorithm can also be computed by a mud algorithm, with comparable space and communication complexity. Our simulation uses Savitch's theorem and therefore has superpolynomial time complexity. We extend our simulation result to some natural classes of approximate and randomized streaming algorithms. We also give negative results, using communication complexity arguments to prove that extensions to private randomness, promise problems and indeterminate functions are impossible. We also introduce an extension of the mud model to multiple keys and multiple rounds.

## 1 Introduction

We now have truly massive data sets, many of which are generated by logging events in physical systems. For example, data sources such as IP traffic logs, web page repositories, search query logs, and retail and financial transactions, consist of billions of items per day, and are accumulated over many days. Internet search companies such as Google, Yahoo!, and MSN, financial companies such as Bloomberg, retail businesses such as Amazon and WalMart, and other companies use this type of data.

In theory, the *data stream model* facilitates the study of algorithms that process such truly massive data sets. Data stream models [1, 9] make one pass over the logs, read and process each item on the stream rapidly and use local storage of size sublinear—typically, polylogarithmic—in the input. There is now a large body of algorithms and lower bounds in data stream models (see [12] for a survey).

Yet, streaming models alone are not sufficient. For example, logs of Internet activity are so large that no single processor can make even a single pass over the data in a reasonable amount of time. Therefore to accomplish even a simple task we need to distribute the computations. This distribution poses numerous challenges, both theoretical and practical. In theory, the streaming model is highly sequential and one needs to design distributed versions of algorithms. In practice, one has to deal with data distribution, synchronization, load balancing, processor failures, etc. Distributed systems such as Google's MapReduce [7] and Apache's Hadoop [4] are successful large scale platforms that can process many terabytes of data at a time, distributed over hundreds or even thousands of machines, and process hundreds of such analyses each day. One reason for their success is that algorithms written for these platforms have a simple form that allow the machines to process the input in an arbitrary order, and combine partial computations using whatever communication pattern is convenient.

The fundamental question that arises is: Does the class of computational tasks supported by these systems differ from the class for which streaming solutions exist? That is, successful though these systems may be in practice, does using multiple machines (rather than a single streaming process) inherently limit the set of possible computations?

To address this problem, we first introduce a simple model for these algorithms, which we refer to as "mud"

*Google, Inc., New York, NY. jonfeld@google.com.

†Google, Inc., New York, NY. muthu@google.com.

‡Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT, Cambridge, MA. tasos@theory.csail.mit.edu. This work was done while visiting Google, Inc., New York, NY.

§Department of IEOR, Columbia University. cliff@ieor.columbia.edu. This work was done while visiting Google, Inc., New York, NY.

¶Department of Computer Science, Cornell University. zoya@cs.cornell.edu. This work was done while visiting Google, Inc., New York, NY.

| | |
|---|---|
| $\Phi(x) = \langle x, x\rangle$ | $\Phi(x) = \langle x, h(x), 1\rangle$ |
| $\oplus(\langle a_1, b_1\rangle, \langle a_2, b_2\rangle) = \langle \min(a_1, a_2), \max(b_1, b_2)\rangle$ | $\oplus(\langle a_1, h(a_1), c_1\rangle, \langle a_2, h(a_2), c_2\rangle) = \begin{cases} \langle a_i, h(a_i), c_i\rangle & \text{if } h(a_i) < h(a_j) \\ \langle a_1, h(a_1), c_1 + c_2\rangle & \text{otherwise} \end{cases}$ |
| $\eta(\langle a, b\rangle) = b - a$ | $\eta(\langle a, b, c\rangle) = a$ if $c = 1$ |

Figure 1: Examples of mud algorithms for computing the total span (left), and a uniform random sample of the unique items in a set (right). Here $h$ is an approximate minwise hash function [5, 6].

(massive, unordered, distributed) algorithms. Later, we relate mud algorithms to streaming computations.

**1.1 Mud algorithms.** Distributed systems such as MapReduce and Hadoop are engines for executing tasks with a certain simple structure over many machines. Algorithms written for these platforms consist of three functions: (1) a *local* function to take a single input data item and output a message, (2) an *aggregation* function to combine pairs of messages, and in some cases (3) a final post-processing step. The system assumes that the local function can be applied to the input data items independently in parallel, and that the aggregation function can be applied to pairs of messages in any order. The platform is therefore able to synchronize the machines very coarsely (assigning them to work on whatever chunk of data becomes available), and does not need machines to share vast amounts of data (thereby eliminating communication bottlenecks)—yielding a highly distributed, robust execution in practice.

**Example.** Consider this simple algorithm to compute the sum of squares of a large set of numbers:[1]

```
x = input_record;
x_squared = x * x;
aggregator: table sum;
emit aggregator <- x_squared;
```

This program is written as if it only runs on a single input record, since it is interpreted as the local function in MapReduce. Instantiating the `aggregator` object as a "table" of type "sum" signals MapReduce to use summation as its aggregation function. "Emitting" `x_squared` into the aggregator defines the message output by the local function. When MapReduce executes this program, the final output is the result of aggregating all the messages (in this case the sum of the squares of the numbers). This output can then be post-processed in some way (e.g., taking the square root, for computing the $L_2$ norm). Many algorithms of this form are used daily for processing logs [15].

**Definition of a mud algorithm.** We now formally define a *mud algorithm* as a triple $m = (\Phi, \oplus, \eta)$. The local function $\Phi : \Sigma \to Q$ maps an input item to a message, the aggregator $\oplus : Q \times Q \to Q$ maps two messages to a single message, and the post-processing operator $\eta : Q \to \Sigma$ produces the final output. The output can depend on the order in which $\oplus$ is applied. Formally, let $\mathcal{T}$ be an arbitrary binary tree circuit with $n$ leaves. We use $m_{\mathcal{T}}(\mathbf{x})$ to denote the $q \in Q$ that results from applying $\oplus$ to the sequence $\Phi(x_1), \ldots, \Phi(x_n)$ along the topology of $\mathcal{T}$ with an arbitrary permutation of these inputs as its leaves. The overall output of the mud algorithm is then $\eta(m_{\mathcal{T}}(\mathbf{x}))$, which is a function $\Sigma^n \to \Sigma$. Notice that $\mathcal{T}$ is *not* part of the algorithm definition, but rather, the algorithm designer needs to make sure that $\eta(m_{\mathcal{T}}(\mathbf{x}))$ is independent of $\mathcal{T}$.[2] We say that a mud algorithm *computes* a function $f$ if $\eta(m_{\mathcal{T}}(\cdot)) = f$ for all trees $\mathcal{T}$.

We give two examples in Figure 1. On the left is a mud algorithm to compute the total span ($\max - \min$) of a set of integers. On the right is a mud algorithm to compute a uniform random sample of the *unique* items in a set (i.e, items that appear at least once) by using an approximate minwise hash function $h$ [5, 6].

The communication complexity of a mud algorithm is $\log |Q|$, the number of bits needed to represent a "message" from one component to the next. We consider the {space, time} complexity of a mud algorithm to be the maximum {space, time} complexity of its component functions $\Phi$, $\oplus$, and $\eta$.[3]

[1] This program is written in Sawzall [15], a language at Google for logs processing that runs on the MapReduce platform. The example is a complete Sawzall program minus some type declarations.

[2] Independence is implied if $\oplus$ is associative and commutative; however, being associative and commutative are not necessary conditions for being independent of $\mathcal{T}$.

[3] This is the only thing that is under the control of the algorithm designer; indeed the actual execution time—which we do not formally define here—will be a function of the number of

**1.2 How do mud algorithms and streaming algorithms compare?** Recall that a mud algorithm to compute a function must work for all computation trees over $\oplus$ operations; now consider the following tree: $\oplus(\oplus(\ldots \oplus (\oplus(q, \Phi(x_1)), \Phi(x_2)), \ldots, \Phi(x_{k-1})), \Phi(x_k))$. This sequential application of $\oplus$ corresponds to the conventional *streaming* model (see e.g. the survey [12]).

Formally, a streaming algorithm is given by $s = (\sigma, \eta)$, where $\sigma : Q \times \Sigma \to Q$ is an operator applied repeatedly to the input stream, and $\eta : Q \to \Sigma$ converts the final state to the output. The notation $s^q(\mathbf{x})$ denotes the state of the streaming algorithm after starting at state $q$, and operating on the sequence $\mathbf{x} = x_1, \ldots, x_k$ *in that order*, that is, $s^q(\mathbf{x}) = \sigma(\sigma(\ldots\sigma(\sigma(q, x_1), x_2), \ldots, x_{k-1}), x_k)$. On input $\mathbf{x} \in \Sigma^n$, the streaming algorithm computes $\eta(s^0(\mathbf{x}))$, where 0 is the starting state. We say a streaming algorithm computes a function $f$ if $f = \eta(s^0(\cdot))$. As in mud, we define the communication complexity to be $\log |Q|$ (which is typically polylogarithmic), and the {space, time} complexity as the maximum {space, time} complexity of $\sigma$ and $\eta$.

If a function can be computed by a mud algorithm, it can also be computed by a streaming algorithm: given a mud algorithm $m = (\Phi, \oplus, \eta)$, there is a streaming algorithm $s = (\sigma, \eta)$ of the same complexity with same output, by setting $\sigma(q, x) = \oplus(q, \Phi(x))$. The central question then is, *can any function computable by a streaming algorithm also be computed by a mud algorithm?* The immediate answer is clearly no. For example, consider a streaming algorithm that counts the number of occurrences of the *first* element in the stream: no mud algorithm can accomplish this since it cannot determine the first element in the input. Therefore, in order to be fair, since mud algorithms work on unordered data, we restrict our attention to functions $\Sigma^n \to \Sigma$ that are *symmetric* (order-invariant) and address this central question.

**1.3 Our Results.** We present the following positive and negative results comparing mud to streaming algorithms, restricted to symmetric functions:

- We show that any deterministic streaming algorithm that computes a symmetric function $\Sigma^n \to \Sigma$ can be simulated by a mud algorithm with the same communication complexity, and the square of its space complexity. This result generalizes to certain approximation algorithms, and randomized algorithms with public randomness.
- We show that the claim above does not extend to richer symmetric function classes, such as when the function comes with a *promise* that the domain is guaranteed to satisfy some property (e.g., finding the diameter of a graph known to be connected), or the function is *indeterminate*, i.e., one of many possible outputs is allowed for "successful computation." (e.g., finding a number in the highest 10% of a set of numbers.) Likewise, with private randomness, the claim above is no longer true.

The simulation in our result takes time $\Omega(2^{\text{polylog}(n)})$ from the use of Savitch's theorem. Therefore our simulation is not a practical solution for executing streaming algorithms on distributed systems; for any specific problem, one may design alternative mud algorithms that are more efficient or even practical. One of the implications of our result however is that any separation between mud algorithms and streaming algorithms for symmetric functions would require lower bounds based on time complexity.

Also, when we consider symmetric problems that have been addressed in the streaming literature, they seem to always yield mud algorithms (e.g., all streaming algorithms that allow insertions and deletions in the stream, or are based on various *sketches* [1] can be seen as mud algorithms). In fact, we are not aware of a specific problem that has a streaming solution, but no mud algorithm with comparable complexity (up to polylog factors in space and per-item time).[4] Our result here provides some insight into this intuitive state of our knowledge and presents rich function classes for which mud is provably as powerful as streaming.

**1.4 Techniques.** One of the core arguments used to prove our positive results comes from an observation in communication complexity. Consider evaluating a symmetric function $f(\mathbf{x})$ given two disjoint portions of the input $\mathbf{x} = \mathbf{x}_A \cdot \mathbf{x}_B$, in each of the two following models. In the *one-way communication model* (OCM), David knows portion $\mathbf{x}_A$, and sends a single message $D(\mathbf{x}_A)$ to Emily who knows portion $\mathbf{x}_B$; she then outputs $E(D(\mathbf{x}_A), \mathbf{x}_B) = f(\mathbf{x}_A \cdot \mathbf{x}_B)$. In the *simultaneous communication model* (SCM) both Alice and Bob send a message $A(\mathbf{x}_A)$ and $B(\mathbf{x}_B)$ respectively, simultaneously to Carol who must compute $C(A(\mathbf{x}_A), B(\mathbf{x}_B)) = f(\mathbf{x}_A \cdot \mathbf{x}_B)$. Clearly, OCM protocols can simulate SCM protocols.[5] At the core, our result relies on observing

machines available, runtime behavior of the platform and these local complexities.

[4] There are specific algorithms—such as one of the algorithms for estimating $F_2$ in [1]—that are sequential and not mud algorithms, but there are other alternative mud algorithms with similar bounds for the problems they solve.

[5] The SCM here is identical to the simultaneous message model [2] or oblivious communication model [16] studied previ-

that SCM protocols can simulate OCMs too, for symmetric functions $f$, by guessing the inputs that result in the particular message received by a party.

To prove our main result—that mud can simulate streaming—we apply the above argument many times over an arbitrary tree topology of $\oplus$ computations, using Savitch's theorem to guess input sequences that match input states of streaming computations. This argument is delicate because we can use the symmetry of $f$ only at the root of the tree; simply iterating the argument at each node in the computation tree independently would yield weaker results that would force the function to be symmetric on *subsets* of the input, which is not assumed by our theorem.

To prove our negative results, we also use communication limitations—of the intermediate SCM. We define order-independent problems easily solved by a single-pass streaming algorithm and then formulate instances that require a polynomial amount of communication in the SCM. The order-independent problems we create are variants of parity and index problems that are traditionally used in communication complexity lower bounds.

**1.5 Multiple rounds and multiple keys.** Mud algorithms model many useful computations performed every day on massive data sets, but to fully capture the capabilities of the modern distributed systems such as MapReduce and Hadoop, we can generalize the algorithms by allowing both multiple keys and multiple rounds. In Section 4 we define this extended model and discuss its computational power.

## 2 Main Result

In this section we give our main result, that any symmetric function computed by a streaming algorithm can also be computed by a mud algorithm.

**2.1 Preliminaries.** As is standard, we fix the space and communication to be polylog$(n)$.[6]

DEFINITION 2.1. *A symmetric function $f : \Sigma^n \to \Sigma$ is in the class MUD if there exists a polylog$(n)$-communication, polylog$(n)$-space mud algorithm $m = (\Phi, \oplus, \eta)$ such that for all $\mathbf{x} \in \Sigma^n$, and all computation trees $\mathcal{T}$, we have $\eta(m_{\mathcal{T}}(\mathbf{x})) = f(\mathbf{x})$.*

ously if there are $k = 2$ players. For $k > 2$, our mud model is not the same as in previous work [2, 16]. The results in [2, 16] as it applies to us are not directly relevant since they only show examples of functions that separate SCM and OCM significantly.

[6]The results in this paper extend to other sub-linear (say $\sqrt{n}$) space, and communication bounds in a natural way.

DEFINITION 2.2. *A symmetric function $f : \Sigma^n \to \Sigma$ is in the class SS if there exists a polylog$(n)$-communication, polylog$(n)$-space streaming algorithm $s = (\sigma, \eta)$ such that for all $\mathbf{x} \in \Sigma^n$ we have $\eta(s^0(\mathbf{x})) = f(\mathbf{x})$.*

Note that for subsequences $\mathbf{x}_\alpha$ and $\mathbf{x}_\beta$, we get $s^q(\mathbf{x}_\alpha \cdot \mathbf{x}_\beta) = s^{s^q(\mathbf{x}_\alpha)}(\mathbf{x}_\beta)$. We can apply this identity to obtain the following simple lemma.

LEMMA 2.1. *Let $\mathbf{x}_\alpha$ and $\mathbf{x}'_\alpha$ be two strings and $q$ a state such that $s^q(\mathbf{x}_\alpha) = s^q(\mathbf{x}'_\alpha)$. Then for any string $\mathbf{x}_\beta$, we have $s^q(\mathbf{x}_\alpha \cdot \mathbf{x}_\beta) = s^q(\mathbf{x}'_\alpha \cdot \mathbf{x}_\beta)$.*

*Proof.* We have $s^q(\mathbf{x}_\alpha \cdot \mathbf{x}_\beta) = s^{s^q(\mathbf{x}_\alpha)}(\mathbf{x}_\beta) = s^{s^q(\mathbf{x}'_\alpha)}(\mathbf{x}_\beta) = s^q(\mathbf{x}'_\alpha \cdot \mathbf{x}_\beta)$ □

Also, note that for some $f \in$ SS, because $f$ is symmetric, the output $\eta(s^0(\mathbf{x}))$ of a streaming algorithm $s = (\sigma, \eta)$ that computes it must be invariant over all permutations of the input; i.e. $\forall x \in \Sigma^n$, permutations $\pi$:

$$\eta(s^0(\mathbf{x})) = f(\mathbf{x}) = f(\pi(\mathbf{x})) = \eta(s^0(\pi(\mathbf{x}))) \tag{2.1}$$

This fact about the *output* of $s$ does not necessarily mean that the *state* of $s$ is permutation-invariant; indeed, consider a streaming algorithm to compute the sum of $n$ numbers that for some reason remembers the first element it sees (which is ultimately ignored by the function $\eta$). In this case the state of $s$ depends on the order of the input, but the final output does not.

**2.2 Statement of the result.** We argued that streaming algorithms can simulate mud algorithms by setting $\sigma(q, x) = \oplus(\Phi(x), x)$, which implies MUD $\subseteq$ SS. The main result in this paper is:

THEOREM 2.1. *For any symmetric function $f : \Sigma^n \to \Sigma$ computed by a $g(n)$-space, $c(n)$-communication streaming algorithm $(\sigma, \eta)$, with $g(n) = \Omega(\log n)$ and $c(n) = \Omega(\log n)$, there exists a $O(c(n))$-communication, $O(g^2(n))$-space mud algorithm $(\Phi, \oplus, \eta)$ that also computes $f$.*

This immediately gives: MUD = SS.

**2.3 Proving Theorem 2.1.** We prove Theorem 2.1 by simulating an arbitrary streaming algorithm with a mud algorithm. The main challenges of the simulation are in

**(i)** achieving polylog communication complexity in the messages sent between $\oplus$ operations,

**(ii)** achieving polylog space complexity for computations needed to support the protocol above, and

**(iii)** extending the methods above to work for an arbitrary computation tree.

We tackle these three challenges in order.

**(i) Communication complexity.** Consider the final application of $\oplus$ (at the root of the tree $\mathcal{T}$) in a mud computation. The inputs to this function are two messages $q_A, q_B \in Q$ that are computed independently from a partition $\mathbf{x}_A, \mathbf{x}_B$ of the input. The output is a state $q_C$ that will lead directly to the overall output $\eta(q_C)$. This task is similar to the one Carol faces in SCM: the input $\Sigma^n$ is split arbitrarily between Alice and Bob, who independently process their input (using unbounded computational resources), but then must transmit only a single symbol from $Q$ to Carol; Carol then performs some final processing (again, unbounded), and outputs an answer in $\Sigma$. We show:

THEOREM 2.2. *Every function $f \in SS$ can be computed in the SCM with communication polylog$(n)$.*

*Proof.* Let $s = (\sigma, \eta)$ be a streaming algorithm that computes $f$. We assume (wlog) that the streaming algorithm $s$ maintains a counter in its state $q \in Q$ indicating the number of input elements it has seen so far.

We compute $f$ in the SCM as follows. Let $\mathbf{x}_A$ and $\mathbf{x}_B$ be the partitions of the input sequence $\mathbf{x}$ sent to Alice and Bob. Alice simply runs the streaming algorithm on her input sequence to produce the state $q_A = s^0(\mathbf{x}_A)$, and sends this to Carol. Similarly, Bob sends $q_B = s^0(\mathbf{x}_B)$ to Carol. Carol receives the states $q_A$ and $q_B$, which contain the sizes $n_A$ and $n_B$ of the input sequences $\mathbf{x}_A$ and $\mathbf{x}_B$. She then finds sequences $\mathbf{x}'_A$ and $\mathbf{x}'_B$ of length $n_A$ and $n_B$ such that $q_A = s^0(\mathbf{x}'_A)$ and $q_B = s^0(\mathbf{x}'_B)$. (Such sequences must exist since $\mathbf{x}_A$ and $\mathbf{x}_B$ are candidates.) Carol then outputs $\eta(s^0(\mathbf{x}'_A \cdot \mathbf{x}'_B))$. To complete the proof:

$$\begin{aligned}
\eta(s^0(\mathbf{x}'_A \cdot \mathbf{x}'_B)) &= \eta(s^0(\mathbf{x}_A \cdot \mathbf{x}'_B)) && \text{(by Lemma 2.1)}\\
&= \eta(s^0(\mathbf{x}'_B \cdot \mathbf{x}_A)) && \text{(by (2.1))}\\
&= \eta(s^0(\mathbf{x}_B \cdot \mathbf{x}_A)) && \text{(by Lemma 2.1)}\\
&= \eta(s^0(\mathbf{x}_A \cdot \mathbf{x}_B)) && \text{(by (2.1))}\\
&= f(\mathbf{x}_A \cdot \mathbf{x}_B) && \text{(correctness of } s)\\
&= f(\mathbf{x}). && \square
\end{aligned}$$

**(ii) Space complexity.** The simulation above uses space linear in the input. We now give a more space-efficient implementation of Carol's computation. More precisely, if the streaming algorithm uses space $g(n)$, we show how Carol can use only space $O(g^2(n))$; this space-efficient simulation will eventually be the algorithm used by $\oplus$ in our mud algorithm.

LEMMA 2.2. *Let $s = (\sigma, \eta)$ be a $g(n)$-space streaming algorithm with $g(n) = \Omega(\log n)$. Then, there is a $O(g^2(n))$-space algorithm that, given states $q_A, q_B \in Q$ and lengths $n_A, n_B \in [n]$, outputs a state $q_C = s^0(\mathbf{x}_C)$, where $\mathbf{x}_C = \mathbf{x}'_A \cdot \mathbf{x}'_B$ for some $\mathbf{x}'_A, \mathbf{x}'_B$ of lengths $n_A, n_B$ such that $s^0(\mathbf{x}'_A) = q_A$ and $s^0(\mathbf{x}'_B) = q_B$. (If such a $q_C$ exists.)*

*Proof.* Note that there may be many $\mathbf{x}'_A, \mathbf{x}'_B$ that satisfy the conditions of the theorem, and thus there are many valid answers for $q_C$. We only require an arbitrary such value. However, if we only have $g^2(n)$ space, and $g^2(n)$ is sublinear, we cannot even write down $\mathbf{x}'_A$ and $\mathbf{x}'_B$. Thus we need to be careful about how we find $q_C$.

Consider a non-deterministic algorithm for computing a valid $q_C$. First, guess the symbols of $\mathbf{x}'_A$ one at a time, simulating the streaming algorithm $s^0(\mathbf{x}'_A)$ on the guess. If after $n_A$ guessed symbols we have $s^0(\mathbf{x}'_A) \neq q_A$, reject this branch. Then, guess the symbols of $\mathbf{x}'_B$, simulating (in parallel) $s^0(\mathbf{x}'_B)$ and $s^{q_A}(\mathbf{x}'_B)$. If after $n_B$ steps we have $s^0(\mathbf{x}'_B) \neq q_B$, reject this branch; otherwise, output $q_C = s^{q_A}(\mathbf{x}'_B)$. This procedure is a non-deterministic, $O(g(n))$-space algorithm for computing a valid $q_C$. By Savitch's theorem [17], it follows that $q_C$ can be computed by a deterministic, $g^2(n)$-space algorithm. (The application of Savitch's theorem in this context amounts to a dynamic program for finding a state $q_C$ such that the streaming algorithm can get from state $q_A$ to $q_C$ and from state 0 to $q_B$ using the same input string of length $n_B$.) $\square$

The running time of this algorithm is super-polynomial from the use of Savitch's theorem, which dominates the running time in our simulation.

**(iii) Finishing the proof for arbitrary computation trees.** To prove Theorem 2.1, we will simulate an arbitrary streaming algorithm with a mud algorithm, setting $\oplus$ to Carol's procedure, as implemented in Lemma 2.2. The remaining challenge is to show that the computation is successful on an arbitrary computation tree; we do this by relying on the symmetry of $f$ and the correctness of Carol's procedure.

*Proof of Theorem 2.1:* Let $f \in$ SS and let $s = (\sigma, \eta)$ be a streaming algorithm that computes $f$. We assume wlog that $s$ includes in its state $q$ the number of inputs it has seen so far. We define a mud algorithm $m = (\Phi, \oplus, \eta)$ where $\Phi(x) = \sigma(0, x)$, and using the same $\eta$ function as $s$ uses. The function $\oplus$, given $q_A, q_B \in Q$ and input sizes $n_A, n_B$, outputs some $q_C = q_A \oplus q_B = s^0(\mathbf{x}_C)$ as in Lemma 2.2. To show the correctness of $m$, we need to show that $\eta(m_{\mathcal{T}}(\mathbf{x})) = f(\mathbf{x})$ for all computation trees $\mathcal{T}$ and all $\mathbf{x} \in \Sigma^n$. For the remainder of the proof, let $\mathcal{T}$ and $\mathbf{x}^* = (x_1^*, \ldots, x_n^*)$ be an arbitrary tree and

input sequence, respectively. The tree $\mathcal{T}$ is a binary in-tree with $n$ leaves. Each node $v$ in the tree outputs a state $q_v \in Q$, including the leaves, which output a state $q_i = \Phi(x_i^*) = \sigma(0, x_i^*) = s^0(x_i^*)$. The root $r$ outputs $q_r$, and so we need to prove that $\eta(q_r) = f(\mathbf{x}^*)$.

The proof is inductive. We associate with each node $v$ a "guess sequence," $\mathbf{x}_v$, which for internal nodes is the sequence $\mathbf{x}_C$ as in Lemma 2.2, and for leaves $i$ is the single symbol $x_i^*$. Note that for all nodes $v$, we have $q_v = s^0(\mathbf{x}_v)$, and the length of $\mathbf{x}_v$ is equal to the number of leaves in the subtree rooted at $v$.

Define a *frontier* of tree nodes to be a set of nodes such that each leaf of the tree has exactly one ancestor in the frontier set. (A node is considered an ancestor of itself.) The root itself is a frontier, as is the complete set of leaves. We say a frontier $V = \{v_1, \ldots, v_k\}$ is *correct* if the streaming algorithm on the data associated with the frontier is correct, that is, $\eta(s^0(\mathbf{x}_{v_1} \cdot \mathbf{x}_{v_2} \cdot \cdots \cdot \mathbf{x}_{v_k})) = f(\mathbf{x}^*)$. Since the guess sequences of a frontier always have total length $n$, the correctness of a frontier set is invariant of how the set is ordered (by (2.1)). Note that the frontier set consisting of all leaves is immediately correct by the correctness of $f$. The correctness of our mud algorithm would follow from the correctness of the root as a frontier set, since at the root, correctness implies $\eta(s^0(\mathbf{x}_r)) = \eta(q_r) = f(\mathbf{x}^*)$.

To prove that the root is a correct frontier, it suffices to define an operation to take an arbitrary correct frontier $V$ with at least two nodes, and produces another correct frontier $V'$ with one fewer node. We can then apply this operation repeatedly until the unique frontier of size one (the root) is obtained. Let $V$ be an arbitrary correct frontier with at least two nodes. We claim that $V$ must contain two children $a, b$ of the same node $c$.[7] To obtain $V'$ we replace $a$ and $b$ by their parent $c$. Clearly $V'$ is a frontier, and so it remains to show that $V'$ is correct. We can write $V$ as $\{a, b, v_1, \ldots, v_k\}$, and so $V' = \{c, v_1, \ldots, v_k\}$. For ease of notation, let $\hat{\mathbf{x}} = \mathbf{x}_{v_1} \cdot \mathbf{x}_{v_2} \cdot \cdots \cdot \mathbf{x}_{v_k}$.

The remainder of the argument follows the logic in the proof of Theorem 2.2.

$$\begin{aligned}
f(\mathbf{x}^*) &= \eta(s^0(\mathbf{x}_a \cdot \mathbf{x}_b \cdot \hat{\mathbf{x}})) && \text{(correctness of } V\text{)} \\
&= \eta(s^0(\mathbf{x}'_a \cdot \mathbf{x}_b \cdot \hat{\mathbf{x}})) && \text{(by Lemma 2.1)} \\
&= \eta(s^0(\mathbf{x}_a \cdot \mathbf{x}'_b \cdot \hat{\mathbf{x}})) && \text{(by (2.1))} \\
&= \eta(s^0(\mathbf{x}'_b \cdot \mathbf{x}'_a \cdot \hat{\mathbf{x}})) && \text{(by Lemma 2.1)} \\
&= \eta(s^0(\mathbf{x}'_a \cdot \mathbf{x}'_b \cdot \hat{\mathbf{x}})) && \text{(by (2.1))} \\
&= \eta(s^0(\mathbf{x}_c \cdot \hat{\mathbf{x}})) && \text{(by Lemma 2.2)} \quad \square
\end{aligned}$$

[7] Proof: consider one of the nodes $a \in V$ furthest from the root. Suppose its sibling $b$ is not in $V$. Then any leaf in the tree rooted at $b$ must have its ancestor in $V$ further from $r$ than $a$; otherwise a leaf in the tree rooted at $a$ would have two ancestors in $V$. This contradicts $a$ being furthest from the root.

Observe that in the above we now have to be careful that the guess for a string is the same length as the original string; this property is guaranteed in Lemma 2.2.

**2.4 Extensions to randomized and approximation algorithms.** We have proved that any deterministic streaming computation of a symmetric function can be simulated by a mud algorithm. However most nontrivial streaming algorithms in the literature rely on randomness, and/or are approximations. Still, our results have interesting implications as described below.

Many streaming algorithms for approximating a function $f$ work by computing some other function $g$ exactly over the stream, and from that obtaining an approximation $\tilde{f}$ to $f$, in postprocessing. For example, sketch-based streaming algorithms maintain counters computed by inner products $c_i = \langle \mathbf{x}, \mathbf{v}_i \rangle$ where $\mathbf{x}$ is the input vector and each $\mathbf{v}_i$ is some vector chosen by the algorithm. From the set of $c_i$'s, the algorithms compute $\tilde{f}$. As long as $g$ is a symmetric function (such as the counters), our simulation results apply to $g$ and hence to the approximation of $f$: such streaming algorithms, approximate though they are, have equivalent mud algorithms. This is a strengthening of Theorem 2.1 to approximations.

Our discussion above can be formalized easily for deterministic algorithms. There are however some details in formalizing it for randomized algorithms. Informally, we focus on the class of randomized streaming algorithms that are order-independent for particular choices of random bits, such as all the randomized sketch-based [1, 10] streaming algorithms. Formally,

DEFINITION 2.3. *A symmetric function $f : \Sigma^n \to \Sigma$ is in the class rSS if there exists a set of polylog(n)-communication, polylog(n)-space streaming algorithms $\{s^R = (\sigma^R, \eta^R)\}_{R \in \{0,1\}^k}$, $k = $ polylog(n), such that for all $\mathbf{x} \in X^n$,*

1. $\Pr_{R \sim \{0,1\}^k} \left[\eta^R(s^R(\mathbf{x})) = f(\mathbf{x})\right] \geq \frac{2}{3}$, *and*
2. *for all $R \in \{0,1\}^k$, and permutations $\pi$, $\eta^R(s^R(\mathbf{x})) = \eta^R(s^R(\pi(\mathbf{x})))$.*

We define the randomized variant of MUD analogously.

DEFINITION 2.4. *A symmetric function $f : \Sigma^n \to \Sigma$ is in rMUD if there exists a set of polylog(n)-communication, polylog(n)-space mud algorithms $\{m^R = (\Phi^R, \oplus^R, \eta^R)\}_{R \in \{0,1\}^k}$, $k = $ polylog(n), such that for all $\mathbf{x} \in X^n$,*

1. *for all computation trees $\mathcal{T}$, we have $\Pr_{R \sim \{0,1\}^k} \left[\eta^R(m_{\mathcal{T}}^R(\mathbf{x})) = f(\mathbf{x})\right] \geq \frac{2}{3}$, and*

2. *for all* $R \in \{0,1\}^k$*, permutations* $\pi$*, and pairs of trees* $\mathcal{T}, \mathcal{T}'$*, we have* $\eta^R(m^R_{\mathcal{T}}(\mathbf{x})) = \eta^R(m^R_{\mathcal{T}'}(\pi(\mathbf{x})))$.

The second property in each of the definitions ensures that each particular algorithm ($s^R$ or $m^R$) computes a deterministic symmetric function after $R$ is chosen. This makes it straightforward to extend Theorem 2.1 to show rMUD = rSS.

## 3 Negative Results

In the previous section, we demonstrated conditions under which mud computations can simulate streaming computations. We saw, explicitly or implicitly, that we have mud algorithms for a function

**(i)** that is total, ie., defined on all inputs,

**(ii)** that has one unique output value, and,

**(iii)** that has a streaming algorithm that, if randomized, uses public randomness.

In this section, we show that each one of these conditions is necessary: if we drop any of them, we can separate mud from streaming. Our separations are based on communication complexity lower bounds in the SCM model, which suffices (see the "communication complexity" paragraph in Section 2.3).

**3.1 Private Randomness.** In the definition of rMUD, we assumed that the same random string $R$ was given to each component; i.e, public randomness. We show that this condition is necessary in order to simulate a randomized streaming algorithm, even for the case of total functions. Formally, we prove:

THEOREM 3.1. *There exists a symmetric total function* $f \in$ *rSS, such that there is no randomized mud algorithm for computing* $f$ *using only private randomness.*

*Proof.* We will demonstrate a total function $f$ that is computable by a single-pass, randomized polylog($n$)-space streaming algorithm, but any SCM protocol for $f$ with private randomness has communication complexity $\Omega(\sqrt{n})$. Our proof uses a reduction from the *string-equality problem* to a problem that we call SETPARITY. In the later problem, we are given a collection of records $S = (i_1, b_1), (i_2, b_2), \ldots, (i_n, b_n)$, where for each $j \in [n]$, we have $i_j \in \{0, \ldots, n-1\}$, and $b_j \in \{0,1\}$. We are asked to compute the following function, which is clearly a total function under a natural encoding of the input:

$$f(S) = \begin{cases} 1 & \text{if } \forall t \in \{0,\ldots,n-1\}, \sum_{j:i_j=t} b_j \bmod 2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

We give a randomized streaming algorithm that computes $f$ using the $\epsilon$-biased generators of [13]. Next, in order to lower-bound the communication complexity of a SCM protocol for SETPARITY, we use the fact that any SCM protocol for string-equality has complexity $\Omega(\sqrt{n})$[3, 14].

A randomized streaming algorithm for computing $f$ works as follows. We pick an $\epsilon$-biased family of $n$ binary random variables $X_0, \ldots, X_{n-1}$, for some $\epsilon < 1/2$. Such a family has the property for any $S \subseteq [n]$,

$$\mathbf{Pr}[\sum_{i \in S} X_i \bmod 2 = 1] > 1/4.$$

Moreover, this family can be constructed using $O(\log n)$ random bits, such that the value of each $X_i$ can be computed in time $\log^{O(1)} n$ [13]. We can thus compute in a streaming fashion the bit $B = b_1 \cdot X_{i_1} + b_2 \cdot X_{i_2} + \ldots + b_n \cdot X_{i_n}$. Observe that if $f(S) = 1$, then $Pr[B = 1] = 0$. On the other hand, if $f(S) = 0$, then let

$$A = \{t \in \{0,\ldots,n-1\} | \sum_{j:i_j=t} b_j \bmod 2 = 1\}.$$

We have $\mathbf{Pr}[B = 1] = \mathbf{Pr}[\sum_{i \in A} X_i \bmod 2 = 1] > 1/4$.

Thus, by repeating in parallel $O(\log n)$ times, we obtain a randomized streaming algorithm for SETPARITY, that succeeds with high probability.

It remains to show that there is no SCM protocol for SETPARITY with communication complexity $o(\sqrt{n})$. We will use a reduction from the string equality problem [3, 14]. Alice gets a string $x_1, \ldots, x_n \in \{0,1\}^n$, and Bob gets a string $y_1, \ldots, y_n \in \{0,1\}^n$. They independently compute the sets of records $S_A = \{(1, x_1), \ldots, (n, x_n)\}$, and $S_B = \{(1, y_1), \ldots, (n, y_n)\}$. It is easy to see that $f(S_A \cup S_B) = 1$ iff the answer to the string-equality problem is YES. Thus, any private-randomness protocol for $f$ has communication complexity $\Omega(\sqrt{n})$. □

**3.2 Promise Functions.** In many cases we would like to compute functions on an input with a particular structure (e.g., a connected graph). Motivated by this, we define the classes pMUD and pSS capturing respectively mud and streaming algorithms for symmetric functions that are not necessarily total (they are defined only on inputs that satisfy a property that is promised).

DEFINITION 3.1. *Let* $A \subseteq \Sigma^n$*. A symmetric function* $f : A \to \Sigma$ *is in the class pMUD if there exists a polylog*($n$)*-communication, polylog*($n$)*-space mud algorithm* $m = (\Phi, \oplus, \eta)$ *such that for all* $\mathbf{x} \in A$*, and computation trees* $\mathcal{T}$*, we have* $\eta(m_{\mathcal{T}}(\mathbf{x})) = f(\mathbf{x})$.

DEFINITION 3.2. *Let* $A \subseteq \Sigma^n$*. A symmetric function* $f : A \to \Sigma$ *is in the class pSS if there exists a polylog*($n$)-

*communication, polylog($n$)-space streaming algorithm $s = (\sigma, \eta)$ such that for all $\mathbf{x} \in A$ we have $s^0(\mathbf{x}) = f(\mathbf{x})$.*

THEOREM 3.2. *pMUD $\subsetneq$ pSS.*

To prove Theorem 3.2, we introduce a promise problem, that we call SYMMETRICINDEX, and show that it is in pSS but not in pMUD. Intuitively, we want to define a problem in which the input will consist of two sets of records. In the first set, we are given a $n$-bit string $x_1, \ldots, x_n$, and a query index $p$. In the second set, we are given a $n$-bit string $y_1, \ldots, y_n$, and a query index $q$. We want to compute either $x_q$, or $y_p$, and we are guaranteed that $x_q = y_p$. Formally, the alphabet of the input is $\Sigma = \{\mathsf{a}, \mathsf{b}\} \times [n] \times \{0,1\} \times [n]$. An input $S \in \Sigma^{2n}$ is some arbitrary permutation of a sequence with the form

$$\begin{aligned} S &= (\mathsf{a}, 1, x_1, p), (\mathsf{a}, 2, x_2, p), \ldots, (\mathsf{a}, n, x_n, p), \\ &\quad (\mathsf{b}, 1, y_1, q), (\mathsf{b}, 2, y_2, q), \ldots, (\mathsf{b}, n, y_n, q). \end{aligned}$$

Additionally, the set $S$ satisfies the promise that $x_q = y_p$. Our task is to compute the function $f(S) = x_q$. We give a deterministic polylog($n$)-space streaming algorithm for SYMMETRICINDEX, and we show that any deterministic SCM protocol for the same problem has communication complexity $\Omega(n)$.

We start by giving a deterministic polylog($n$)-space streaming algorithm for SYMMETRICINDEX that implies SYMMETRICINDEX $\in$ pSS. The algorithm is given the elements of $S$ in an arbitrary order. If the first record is $(\mathsf{a}, i, x_i, p)$ for some $i$, the algorithm streams over the remaining records until it gets the record $(\mathsf{b}, p, y_p, q)$ and outputs $y_p$. If the first record is $(\mathsf{b}, j, y_j, q)$ for some $j$, then the algorithm streams over the remaining records until it gets the record $(\mathsf{a}, q, x_q, p)$. In either case we output $x_q = y_p$.

We next show that SYMMETRICINDEX $\notin$ pMUD. It suffices to show that any deterministic SCM protocol for SYMMETRICINDEX requires $\Omega(n)$ bits of communication. Consider such a protocol in which Alice and Bob each send $b$ bits to Carol, and assume for the sake of contradiction that $b < n/40$. Let $I$ be the set of instances to the SYMMETRICINDEX problem. Simple counting yields that $|I| = n^2 2^{2n-1}$. For an instance $\phi \in I$, we split it into two pieces $\phi_A$, for Alice and $\phi_B$, for Bob. We assume that these pieces are

$$\begin{aligned} \phi_A &= (\mathsf{a}, 1, x_1^\phi, p^\phi), \ldots, (\mathsf{a}, n, x_n^\phi, p^\phi), \text{ and} \\ \phi_B &= (\mathsf{b}, 1, y_1^\phi, q^\phi), \ldots, (\mathsf{b}, n, y_n^\phi, q^\phi). \end{aligned}$$

For this partition of the input, let $I_A$ and $I_B$ be the sets of possible inputs of Alice, and Bob respectively. Alice computes a function $h_A : I_A \to [2^b]$, Bob computes a function $h_B : I_B \to [2^b]$, and each sends the result to Carol. Intuitively, we want to argue that if Alice sends at most $n/40$ bits to Carol, then for an input that is chosen uniformly at random from $I$, Carol does not learn the value of $x_i$ for at least some large fraction of the indices $i$. We formalize the above intuition with the following lemma:

LEMMA 3.1. *If we pick $\phi \in I$, and $i \in [n]$ uniformly at random and independently, then:*

- *With probability at least $4/5$, there exists $\chi \neq \phi \in I$, such that $h_A(\phi_A) = h_A(\chi_A)$, $p^\phi = p^\chi$, and $x_i^\phi \neq x_i^\chi$.*
- *With probability at least $4/5$, there exists $\psi \neq \phi \in I$, such that $h_B(\phi_B) = h_B(\psi_B)$, $q^\phi = q^\psi$, and $y_i^\phi \neq y_i^\psi$.*

*Proof.* Because of the symmetry between the cases for Alice and Bob, it suffices to prove the assertion for Alice. For $j \in [2^b]$, $r \in [n]$, let

$$C_{j,r} = \{\gamma \in I | h_A(\gamma_A) = j \text{ and } p^\gamma = r\}.$$

Let $\alpha_{j,r}$ be the set of indices $t \in [n]$, such that $x_t^\gamma$ is fixed, for all $\gamma \in C_{j,r}$. That is,

$$\alpha_{j,r} = \{t \in [n] | \text{ for all } \gamma, \gamma' \in C_{j,r}, x_t^\gamma = x_t^{\gamma'}\}.$$

If we fix $|\alpha_{j,r}|$ elements $x_i$ in all the instances in $C_{j,r}$, then any pair $\gamma, \gamma' \in C_{j,r}$ can differ only in some $x_i$, with $i \notin \alpha_{j,r}$, or in the index $q$, or in $y_t$, with the constraint that $x_q = y_p$. Thus, for each $j, r \in [2^b]$,

$$|C_{j,r}| \leq n \cdot 2^{2n - |\alpha_{j,r}| - 1}. \tag{3.2}$$

Thus, if $|\alpha_{j,r}| \geq n/20$, then $|C_{j,r}| \leq n 2^{39n/20 - 1}$. Pick $\phi \in I$, and $i \in [n]$ uniformly at random, and independently, and let $\mathcal{E}$ be the event that there exists $\chi \neq \phi \in I$, such that $h_A(\phi_A) = h_A(\chi_A)$, $p^\phi = p^\chi$, and $x_i^\phi \neq x_i^\chi$. Then

$$\begin{aligned} \mathbf{Pr}[\mathcal{E}] &= 1 - \frac{\sum_{j \in [2^b], r \in [n]} |C_{j,r}| \cdot |\alpha_{j,r}|}{n \cdot |I|} \\ &\geq 1 - \frac{\sum_{j \in [2^b], r \in [n]} n \cdot 2^{n\frac{39}{20} - 1} \cdot n}{n^3 \cdot 2^{2n-1}} \\ &\quad - \frac{\sum_{j \in [2^b], r \in [n]} |C_{j,r}| \cdot n/20}{n^3 \cdot 2^{2n-1}} \\ &\geq 1 - \frac{2^{n/40} \cdot n^3 \cdot 2^{n\frac{39}{20} - 1} + n^2 \cdot 2^{2n-1} \cdot n/20}{n^3 \cdot 2^{2n-1}} \\ &> 4/5, \end{aligned}$$

for sufficiently large $n$. □

Consider an instance $\phi$ chosen uniformly at random from $I$. Clearly, $p^\phi$, and $q^\phi$ are distributed uniformly in $[n]$, $q^\phi$, and $\phi_A$ are independent, and $p^\phi$, and $\phi_B$ are independent. Thus, by Lemma 3.1 with probability at least $1 - 2\left(\frac{1}{5}\right)$ there exist $\chi, \psi \in I$, such that:

- $h_A(\phi_A) = h_A(\chi_A)$, $p^\phi = p^\chi$, and $x^\phi_{q^\phi} \neq x^\chi_{q^\phi}$.
- $h_B(\phi_B) = h_B(\psi_B)$, $q^\phi = q^\psi$, and $y^\phi_{p^\phi} \neq y^\psi_{p^\phi}$.

Consider now the instance $\gamma = \chi_A \cup \psi_B$. That is,

$$\begin{aligned}\gamma \;=\; & (\mathtt{a}, 1, x^\chi_1, p^\chi), \ldots, (\mathtt{a}, n, x^\chi_n, p^\chi), \\ & (\mathtt{b}, 1, y^\psi_1, q^\psi), \ldots, (\mathtt{b}, n, y^\psi_n, q^\psi)\end{aligned}$$

Observe that

$$\begin{aligned} x^\gamma_{q^\gamma} &= x^\chi_{q^\psi} && \text{(by the definition of } \gamma) \\ &= x^\chi_{q^\phi} = 1 - x^\phi_{q^\phi} \\ &= 1 - y^\phi_{p^\phi} && \text{(by the promise for } \phi) \\ &= y^\psi_{p^\phi} = y^\psi_{p^\chi} \\ &= y^\gamma_{p^\gamma} && \text{(by the definition of } \gamma). \end{aligned}$$

Thus, $\gamma$ satisfies the promise of the problem (i.e., $\gamma \in I$). Moreover, we have $h_C(h_A(\phi^A), h_B(\phi^B)) = h_C(h_A(\gamma^A), h_B(\gamma^B))$, while $x^\phi_{q^\phi} \neq x^\gamma_{q^\gamma}$. It follows that the protocol is not correct. We have thus shown that pMUD $\subsetneq$ pSS and proved Theorem 3.2.

**3.3 Indeterminate Functions.** In some applications, the function we wish to compute may have more than one "correct" answer. We define the classes iMUD and iSS to capture the computation of "indeterminate" functions.

DEFINITION 3.3. *A total symmetric function $f : \Sigma^n \to 2^\Sigma$ is in the class iMUD if there exists a polylog$(n)$-communication, polylog$(n)$-space mud algorithm $m = (\Phi, \oplus, \eta)$ such that for all $\mathbf{x} \in \Sigma^n$, and computation trees $\mathcal{T}$, we have $\eta(m_\mathcal{T}(\mathbf{x})) \in f(\mathbf{x})$.*

DEFINITION 3.4. *A total symmetric function $f : \Sigma^n \to 2^\Sigma$ is in the class iSS if there exists a polylog$(n)$-communication, polylog$(n)$-space streaming algorithm $s = (\sigma, \eta)$ such that for all $\mathbf{x} \in X^n$ we have $s^0(\mathbf{x}) \in f(\mathbf{x})$.*

Consider a promise function $f : A \to \Sigma$, such that $f \in$ pMUD. We can define a total indeterminate function $f' : \Sigma^n \to 2^\Sigma$, such that for each $x \in A$, $f'(x) = f(x)$, and for each $x \notin A$, $f(x) = \Sigma$. That is, for any input that satisfies the promise of $f$, the two functions are equal, while for all other inputs, any output is acceptable for $f'$. Clearly, a streaming or mud algorithm for $f'$, is also a streaming or mud algorithm for $f$ respectively. Therefore, Theorem 3.2 implies the following result.

THEOREM 3.3. *iMUD $\subsetneq$ iSS.*

## 4 Multiple Keys, Multiple Passes

The MUD class includes many useful computations performed every day on massive data sets, but to fully capture the capabilities of the modern distributed systems such as MapReduce and Hadoop, we can generalize it in two different ways.

First, we can allow multiple mud algorithms running simultaneously over the same input. This is implemented by computing (*key*, *value*) pairs for each input $x_i$, and then aggregating the values with the same key using the $\oplus$ function. More formally, a *multi-key* mud algorithm is a triple $(\Phi, \oplus, \eta)$ where $\Phi : \Sigma \to 2^{K \times Q}$, $K$ is the set of keys, and $\oplus$ and $\eta$ are defined as in single-key mud algorithms (for each key). The communication complexity of the multi-key mud algorithm is $\log |Q|$ per key. As before, we consider the {space, time} complexity of a multi-key mud algorithm to be the maximum {space, time} complexity of its component functions $\Phi$, $\oplus$, and $\eta$ per key. When the algorithm is executed on the input $\mathbf{x}$, the function $\Phi$ produces a set $\cup_i \Phi(x_i)$ of key-value pairs. Each set of values with the same key is aggregated independently using $\oplus$ and an arbitrary computation tree, followed by a final application of $\eta$. The final output is an unordered set of symbols $\mathbf{x}' \in \Sigma^{n'}$, where $n'$ is the number of unique keys produced by $\Phi$. For more details on how this is achieved in a practical system, see [4, 7].

Second, we can allow multiple rounds of computation, where each round is a mud algorithm, perhaps using multiple keys. Since each round constitutes a function $\Sigma^n \to \Sigma^{n'}$, mud algorithms naturally compose to produce an overall function $\Sigma^n \to \Sigma^{n'}$.

**Example.** Let $\mathbf{x} \in [m]^n$, and define $n_i$ to be the number of occurrences of the element $i$ in the sequence $\mathbf{x}$. The *$k$-th frequency* moment of $\mathbf{x}$ is the quantity $F_k(\mathbf{x}) = \sum_{i \in [m]} n_i^k$. For any constant $k$, the function $F_k(\mathbf{x})$ can be computed with an $m$-key, 2-round mud algorithm as follows: (1) In the first pass we compute the frequencies $\{n_i\}_{i \in [m]}$ using the element names as keys, and counting with $\oplus$. (2) In the second pass, we just need to compute $\sum_{i \in [m]} n_i^k$. We do this with a single-key mud algorithm where $\Phi(x) = x^k$, and $\oplus$ is addition. This is because the generalized mud algorithms can use polylog$(n)$ bits of communication *per* key per round. In contrast, one-pass streaming algorithms cannot even approximate $F_k$ for certain $k$ with polylog$(n)$ space [1]. □

These extensions make the model much more powerful. In fact, one can solve any problem in NC [8] with an $O(n)$-key, polylog$(n)$-round mud algorithm (we leave

the full statement and proof of this result for a full version of the paper).

## 5 Concluding Remarks

Conventional streaming algorithms that make a pass over data with a single processor are insufficient for large-scale data processing tasks. Modern distributed systems like Google's MapReduce [7] and Apache's Hadoop [4] rely on massive, unordered, distributed (mud) computations to do data analysis in practice, and obtain speedups. We have introduced mud algorithms, and asked how the power of these algorithms compares to conventional streaming. Our main result is that any symmetric function that can be computed by a streaming algorithm can also be computed by a mud algorithm with comparable space and communication resources, showing the equivalence of the two classes in principle. At the heart of the proof is a nondeterministic simulation of a streaming algorithm that guesses the stream, and an application of Savitch's theorem to be space-efficient. This result formalizes some of the intuition that has been used in designing streaming algorithms in the past decade. This result has certain natural extensions to approximate and randomized computations, and we show that other natural extensions to richer classes of symmetric functions are impossible.

Unfortunately, our simulation does not immediately provide a practical algorithm for obtaining speedups from distributing streaming computations over multiple machines because of the running time needed for the simulation, and for any specific streaming computation, alternative mud algorithms may be faster. This raises the following question: Can one obtain a more time-efficient simulation for Theorem 2.1? Another interesting question, posed by D. Sivakumar [11], is whether there are natural problems for which this simulation provides an interesting algorithm.

**Beyond One-pass Streaming.** In the past decade, researchers have generalized single pass streaming to multiple passes and to semi-streaming, where one has a linear number of streaming computations. Here we offer a definition of a multiple-key, multiple-pass mud algorithm that extends the mud model analogously. We hope this will inspire further work in this area to develop the theoretical foundation for successful modern distributed systems.


## Acknowledgements

We thank the anonymous referees for several suggestions to improve a previous version of this paper, and for suggesting the use of $\epsilon$-biased generators. We also thank Sudipto Guha and D. Sivakumar for helpful discussions.


## References


[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *STOC*, pages 20–29, 1996.

[2] L. Babai, A. Gal, P. Kimmel, and S. Lokam. Simultaneous messages and communication. *Univ of Chicago, Technical Report*, 1996.

[3] L. Babai and P. G. Kimmel. Randomized simultaneous messages: Solution of a problem of yao in communication complexity. In *Computational Complexity*, page 239, Washington, DC, USA, 1997. IEEE Computer Society.

[4] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware, 2005. Wiki at `http://lucene.apache.org/hadoop/`.

[5] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci. 60(3)*, pages 630–659, 2000.

[6] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. *ESA*, pages 323–334, 2002.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[8] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[9] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *Technical Note 1998-011, Digital Systems Research Center, Palo Alto, CA*, 1998.

[10] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of ACM*, pages 307–323, 2006.

[11] A. McGregor. Open problems in data streams research. `http://www.cse.iitk.ac.in/users/sganguly/data-stream-probs.pdf`.

[12] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 2005.

[13] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM Journal on Computing*, 22(4):838–856, August 1993.

[14] I. Newman and M. Szegedy. Public vs. private coin flips in one round communication games (extended abstract). In *STOC*, pages 561–570, New York, NY, 1996. ACM Press.

[15] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):227–298, 2005.

[16] P. Pudlak, V. Rodl, and J. Sgall. Boolean circuits, tensor ranks and communication complexity. *Manuscript*, 1994.

[17] Savitch. Maze recognizing automata and nondeterministic tape complexity. *Journal of Computer and System Sciences*, 1973.