In-situ MapReduce for Log Processing

Dionysios Logothetis, Chris Trezzo*, Kevin C. Webb, and Kenneth Yocum
*UCSD Department of Computer Science, *Salesforce.com, Inc.*

Abstract

Log analytics are a bedrock component of running many of today's Internet sites. Application and click logs form the basis for tracking and analyzing customer behaviors and preferences, and they form the basic inputs to ad-targeting algorithms. Logs are also critical for performance and security monitoring, debugging, and optimizing the large compute infrastructures that make up the compute "cloud", thousands of machines spanning multiple data centers. With current log generation rates on the order of 1–10 MB/s per machine, a single data center can create tens of TBs of log data a day.

While bulk data processing has proven to be an essential tool for log processing, current practice transfers all logs to a centralized compute cluster. This not only consumes large amounts of network and disk bandwidth, but also delays the completion of time-sensitive analytics. We present an in-situ MapReduce architecture that mines data "on location", bypassing the cost and wait time of this store-first-query-later approach. Unlike current approaches, our architecture explicitly supports reduced data fidelity, allowing users to annotate queries with latency and fidelity requirements. This approach fills an important gap in current bulk processing systems, allowing users to trade potential decreases in data fidelity for improved response times or reduced load on end systems. We report on the design and implementation of our in-situ MapReduce architecture, and illustrate how it improves our ability to accommodate increasing log generation rates.

1 Introduction

Scalable log processing is a crucial facility for running large-scale Internet sites and services. Internet firms process click logs to provide high-fidelity ad targeting, system and network logs to determine system health, and application logs to ascertain delivered service quality. For instance, E-commerce and credit card companies analyze point-of-sales transactions for fraud detection, while infrastructure providers use log data to detect hardware misconfigurations and load-balance across data centers [6, 30].

This semi-structured log data is produced across one or more data centers that contain thousands of machines. It is not uncommon for such machines to produce data at rates of 1–10 MB/s [4]. Even at the low end (1 MB/s), a modest 1000-node cluster could generate 86 TB of raw logs in a single day. To handle these large data sets, many sites use data parallel processing systems like MapReduce [12] or Dryad [20]. Such frameworks allow businesses to capitalize on cheap hardware, harnessing thousands of commodity machines to process enormous data sets.

The dominant approach is to move the data to a single cluster dedicated to running such a bulk processing system. In this "store-first-query-later" approach [13] users load data into a distributed file system and then execute queries.[1] For example, companies like Facebook and Rackspace analyze tens of terabytes of log data a day by pulling the data from hundreds to thousands of machines, loading it into HDFS (the Hadoop Distributed File System), and then running a variety of MapReduce jobs on a large Hadoop cluster [17]. Many of the processing jobs are time sensitive, with sites needing to process logs in 24 hours or less, enabling accurate user activity models for re-targeting advertisements, fast social network site updates, or up-to-date mail spam and usage statistics.

However, this centralized approach to log processing has two drawbacks. First, it fundamentally limits its scale and timeliness. For example, to sink 86 TB of log data in less than an hour (48 minutes) would require 300 Gb/s of dedicated network and disk bandwidth. This limits processing on the MapReduce cluster as the transfer occupies disk arms, and places a large burden on the data

[1] Here we consider queries as single or related MapReduce jobs.

center network, even if well provisioned. Second, the approach must sacrifice availability or blindly return incomplete results in the presence of heavy server load or failures. Current bulk processing systems provide strict consistency, failing if not all data is processed. This implies that either users delay processing until logs are completely delivered or that their analytics run on incomplete data.

In fact, though, one does not have to make this either-or choice. It is often possible to accurately summarize or extract useful information from a subset of log data, as long as we have a systematic method for characterizing data fidelity. For example, if a user can ascertain whether a particular subset of log data is a uniform sampling, one can capture the relative frequency of events (e.g., failures or user clicks) across server logs.

To meet these goals we present an "in-situ" MapReduce (iMR) architecture for moving analytics on to the log servers themselves. By transforming the data in place, we can reduce the volume of data crossing the network and the time to transform and load the data into stable distributed storage. However, this processing environment differs significantly from a dedicated Hadoop cluster. Nodes are not assumed to share a distributed file system, implying that data is not replicated nor available at other nodes. And the servers are not dedicated to log processing; they must also support client-facing requests (web front ends, application servers, databases, etc.). Thus unlike traditional MapReduce architectures, our in-situ approach accepts that data may naturally be unavailable either because of failures or because there are insufficient resources to meet latency requirements.

This work makes the following contributions:

- **Continuous MapReduce model:** Unlike batch-oriented workloads, log analytics take as input essentially infinite input streams. iMR supports an extended MapReduce programming model that allows users to define continuous MapReduce jobs with sliding/tumbling windows [7]. This allows incremental updates, re-using prior computation when data arrives/departs. Because iMR directly supports stream processing, it can run standard MR jobs continuously without modification.
- **Lossy MapReduce processing:** iMR supports lossy MapReduce processing to increase result availability when sourcing logs from thousands of servers. To interpret partial results, we present C^2, a metric of result quality that takes into account the spatial and temporal nature of log processing. In iMR users may set a target C^2 for acceptable result fidelity, allowing the system to process a subset of the data to decrease latency, avoid excessive load on the log servers, or accommodate node or network failures.
- **Architectural lessons:** We explore the iMR architecture with a prototype system based on a best-effort distributed stream processor, Mortar [22]. We develop efficient strategies for internally grouping key-value pairs in the network using sub-windows or *panes*, and explore the impact of failures on result fidelity and latency. We also develop load cancellation and shedding policies that allow iMR to maximize result quality when there are insufficient server resources to provide perfect results.

Section 2 gives an overview of the system design, discusses related work, and describes how iMR performs continuous MapReduce processing using windows. Section 3 introduces our notion of result quality C^2, useful ways to express C^2, and how the system efficiently maintains that metric. Section 4 discusses our modifications to Mortar to support iMR. We evaluate the system in Section 5, looking at system scalability, load shedding, and data fidelity control. In particular we explore how C^2 affects results when extracting simple count statistics, performing click-stream analysis, and building an HDFS anomaly detector.

2 Design overview

iMR is designed to complement, not replace traditional cluster-based architectures. It is meant for jobs that filter or transform log data either for immediate use or before loading it into a distributed storage system (e.g., HDFS) for follow-on analysis. Moreover, today's batch processing queries exhibit characteristics that make them amenable to continuous, in-network processing. For instance, many analytics are highly selective. A 3-month trace from a Microsoft large-scale data processing system showed that filters were often highly selective (17 - 26%) [16], and the first step for many Facebook log analytics is to reduce the log data by 80% [4]. Additionally, many of these queries are update-driven, integrate the most recent data arrivals, and recur on an hourly, daily, or weekly basis.

Below we summarize how in-situ MapReduce ensures that log processing is:

Scalable: The target operating environment consists of thousands of servers in one or more data centers, each producing KBs to MBs of log data per second. In iMR, MapReduce jobs run continuously on the servers themselves (shown on the right in Figure 1). This provides horizontal scaling by simply running in-place, i.e, the processing node count is proportional to the number of data sources. This design also lowers the cost and latency of loading data into a storage cluster by filtering data on site and using in-network aggregation, if the user's reduce implements an aggregate function [14].

Responsive: Today the latency of log analytics dictates various aspects of a site's performance, such as the

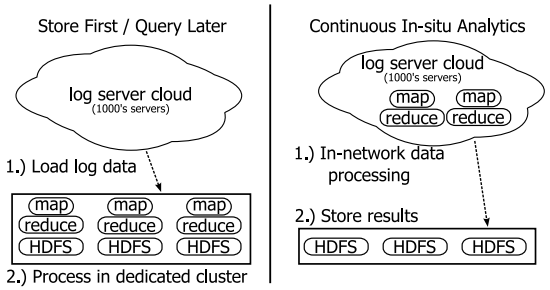


Figure 1: The in-situ MapReduce architecture avoids the cost and latency of the store-first-query-later design by moving processing onto the data sources.

speed of social network updates or accuracy of ad targeting. The in-situ MapReduce (iMR) architecture builds on previous work in stream processing [5, 7, 9] to support low-latency continuous log processing. Like stream processors, iMR MapReduce jobs can process over sliding windows, updating and delivering results as new data arrives.

Available: iMR's lossy data model allows the system to return results that may be incomplete. This allows the system to improve result availability in the event of failures or processing and network delays. Additionally, iMR may pro-actively reduce processing fidelity through load shedding, reducing the impact on existing server tasks. iMR attaches a metric of result quality to each output, allowing users to judge the relative accuracy of processing. Users may also explicitly trade fidelity for improved result latency by specifying latency and fidelity bounds on their queries.

Efficient: A log processing architecture should make parsimonious use of computational and network resources. iMR explores the use of sub-windows or *panes* for efficient continuous processing. Instead of recomputing each window from scratch, iMR allows incremental processing, merging recent data with previously computed panes to create the next result. And adaptive load-shedding policies ensure that nodes use compute cycles for results that meet latency requirements.

Compatible: iMR supports the traditional MapReduce API, making it trivial to "port" existing MapReduce jobs to run in-situ. It provides a single extension, *uncombine*, to allow users to further optimize incremental processing in some contexts (Section 2.3.2).

2.1 In-situ MapReduce jobs

A MapReduce job in iMR is nearly identical to that in traditional MapReduce architectures [12]. Programmers specify two data processing functions: map and reduce. The map function outputs key-value pairs, $\{k, v\}$, for each input record, and the reduce processes each group of values, $v[]$, that share the same key k. iMR is designed for queries that are either highly selective or employ reduce functions that are distributive or algebraic aggregates [14]. Thus we expect that users will also specify the MapReduce *combiner*, allowing the underlying system to merge values of a single key to reduce data movement and distribute processing overhead. The use of a combiner allows iMR to process windows incrementally and further reduce data volumes through in-network aggregation. The only non-standard (but optional) function iMR MapReduce jobs may implement is *uncombine*, which we describe in Section 2.3.2.

However, the primary way in which iMR jobs differ is that they emit a stream of results computed over continuous input, e.g., server log files. Like data stream processors [7], iMR bounds computation over these (perhaps infinite) data streams by processing over a *window* of data. The window's *range* R defines the amount of data processed in each result, while the window's *slide* S defines its update frequency. For example, a user could count error events over the last 24 hours of log records ($R = 24\,\texttt{hours}$), and update the count every hour ($S = 1\,\texttt{hour}$). This *sliding* window, one whose slide S is less than its range R, may be in terms of wall-clock time or logical index, such as record count, bytes, or any user-defined sequence number. Users specify R and S with simple annotations to the reduce function.

While sufficient for real-time log processing, a MapReduce job in iMR may reference historical log data as well. Doing so requires a job-level annotation that specifies the point in the local log to *begin* B and the total data to consume, the *extent* E. If unspecified, the job continues to process, possibly catching up to real-time processing.

2.2 Job execution

In general, MapReduce architectures have three primary tasks: the parallel execution of the map phase, grouping input records by key, and the parallel execution of the reduce phase. In cluster-based MapReduce systems, like Hadoop, each map task produces key-value pairs, $\{k,v\}$, from raw input records at individual nodes in the cluster. The map tasks then group values by their key k, and split the set of keys into r partitions. After the map tasks finish, the system starts a reduce task for each partition r. These tasks first download their partition's key-value pairs from each mapper (the *shuffle*), finish grouping values, and then call reduce once for every $\{k,v[]\}$ pair.

iMR distributes the work of a MapReduce job across multiple trees, one for each reducer partition. Figure 2 illustrates one such tree; iMR co-locates map processing on the server nodes themselves, sourcing input records

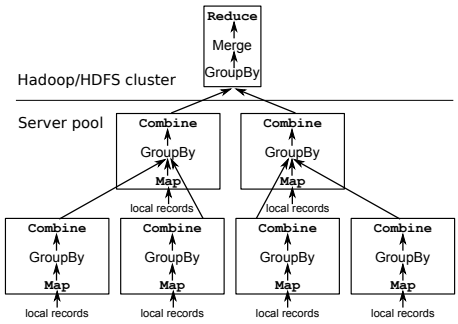


Figure 2: This illustrates the physical instantiation of one iMR MapReduce partition as a multi-level aggregation tree.

(tuples) from the local node's log file. The dedicated processing cluster hosts the root, which executes the user's reduce function. This tree uses the combine API to aggregate intermediate data at every mapper in a manner similar to traditional MapReduce architectures. However, like Dryad [32], iMR can use multi-level aggregation trees to further reduce the data crossing the network.

In general, this requires aggregate or *decomposable* functions that can be computed incrementally [15, 23, 32]. Here we are interested in two broad categories of aggregate functions [21]. *Holistic* aggregates require partial values whose size is in proportion to their input data, e.g., `union`, `median` or `groupby`. In contrast, *bounded* aggregates have constant-sized partial values, e.g., `sum` or `max`, and present the greatest opportunities for data reduction.

2.3 Window processing with panes

iMR supports sliding processing windows not just because they bound computation on infinite streams, but because they also enable incremental computations. However, they do not immediately lend themselves to efficient in-network processing. Consider a simple aggregation strategy where each log server accumulates all key-value pairs for each logical window and nodes in the aggregation tree combine these entire windows.

We can see that this strategy isn't efficient for our example sliding window query. In this case, every event record would be included in 24 successive results. Thus every input key-value pair in a sliding window would be grouped, combined, and transmitted for each update (slide) of the window or R/S times. To reduce these overheads, iMR adapts the use of sub-windows or *panes* to efficiently compute aggregates over sliding windows. While the concept of panes was introduced in prior work for single-node stream processors [21]; here we adapt them to distributed in-situ MapReduce processing.

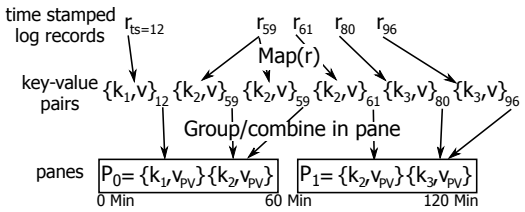


Figure 3: iMR nodes process local log files to produce sub-windows or panes. The system assumes log records have a logical timestamp and arrive in order.

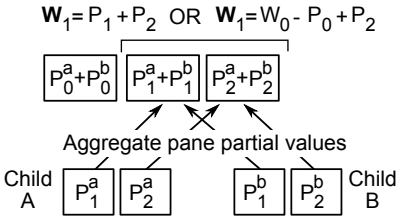


Figure 4: iMR aggregates individual panes P_i in the network. To produce a result, the root may either combine the constituent panes or update the prior window by removing an expired pane and adding the most recent.

2.3.1 Pane management

Panes break a window into multiple equal-sized sub-windows, allowing the system to group and combine key-value records once per sub-window. Nodes in the system generate panes and send them to their parents in the aggregation tree. Thus in iMR, interior nodes in a tree aggregate panes and the root node combines them into each window result. This supports the fundamental grouping operation underlying reduce, a holistic aggregate. By sending panes, rather than sending the entire window up the tree, the system sends a single copy of a key's value, reducing network traffic. Additionally, issuing values at the granularity of panes gives the system fine-grain control on fidelity and load shedding (Section 3.4). It is also the granularity at which failed nodes restart processing, minimizing the gap of dropped data (Section 4.4.2).

Figure 3 illustrates how a single node creates panes from a stream of local log records. Typically, we set the pane size equal to the slide S, though it may be any common divisor of R and S, and each node maintains a sequence of pane partial values P_i. This example uses a processing window with a slide of 60 minutes. When log records first enter the system, iMR tags each one with a non-decreasing user-defined timestamp. The system then feeds these records to the user's map function. After mapping, the system assigns key-value pairs

to each pane, where they are grouped/combined. Note that a pane is complete when a log entry arrives for the following pane (log entries are assumed to be in order).

2.3.2 Window creation

In iMR, the root of each reducer partition must group/combine all keys in the window before executing the user's reduce function and computing the result. Figure 4 illustrates two strategies the root may employ to do so. Here two log servers A and B create panes P_1 and P_2 and send them to the root. The root first groups (and possibly combines) panes with the same index.

The first strategy leverages panes to allow incremental processing with the traditional MapReduce API. The strategy simply uses the existing combine API to merge adjoining panes. In this example each window consists of two panes and W_1 may be constructed by merging P_1^{a+b} with P_2^{a+b}. This improves efficiency by having each overlapping window re-use a pane's partial value; merging window panes is cheaper than repeatedly combining the raw mapped values for each window. This benefit increases with the number of values per key.

However, for sliding windows it is sometimes more efficient to *remove* expired data and then add new data to the prior W. For instance, consider our 24 hour query that updates every hour. In this case the root must combine 24 panes to produce each window. In contrast, the root could remove and add a pane's worth of keys to the prior window W, greatly reducing the volume of keys touched. Assuming that the cost of removing and adding keys to W is equivalent, this strategy is always more efficient than merging all constituent panes when the slide is less than half the range. This requires "differential" [21] functions, i.e. aggregates that are commutative/associative under removals as well as additions. iMR only uses an uncombine strategy when the slide is less than half the range and a user supplies an uncombiner.

3 Lossy MapReduce processing

This section describes the features of iMR that allow it to accommodate data loss. As described earlier, data loss may occur because of node or network failures, or as a consequence of result latency requirements. In such cases, an iMR job may need to report a result before the system has had time to process all the data in the window. The key challenges we address here are a.) how to represent and calculate result quality to allow users to interpret partial results, and b.) how to use this metric to trade result fidelity for improved result latency.

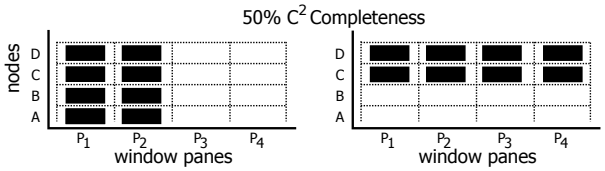


Figure 5: C^2 completeness describes the set of panes each log server contributes to the window. Here we show two different ways in which C^2 represents 50% of the data area: all the nodes process half the data or half the nodes process all the data.

3.1 Measuring data fidelity

A good measure of data fidelity should inform users not only that data is missing, but allows users to ascertain the impact of data loss on query accuracy. One measure of result quality used for in-network aggregates is *completeness*, the number or fraction of nodes whose data is represented in the final answer [22, 25]. Alternatively, systems like Hadoop Online (HOP) output partial answers as data arrives, and annotate them with *progress*, the percent of total data processed. Unfortunately, neither metric is sufficiently descriptive for window-based processing. Completeness cannot differentiate between a single node that produces log records that span the entire window and a node that does not. Similarly, a simple progress metric fails to account for the source of processed data.

Here we present a completeness metric, C^2, that leverages the natural distribution of log data across both space (log server nodes) and time (the window range). C^2 represents the data *area* included in the final result as the number of unique data panes that have been successfully integrated into the window. Logically, the root maintains C^2 like a scoreboard, with a mark for every successfully received pane in the window (Figure 5). Thus C^2 tracks the set of nodes whose log data contributed to the window, as well as how that log data was distributed across the result window. iMR can summarize this raw information as independent percentages of temporal (x-axis) and spatial (y-axis) completeness or simply as an area, the total result coverage.

Figure 5 illustrates how C^2 may reflect two different scenarios that process the same data area (in this case C^2 =50%). In the first case, all the nodes process half the data and in the second, half the nodes process all the data. There are, of course, other scenarios where the product of the percentage of nodes and percentage of the window processed will be 50%, and C^2 allows users to differentiate between them. Note that C^2 explains what was included in the result, not what was *missing*, which is a much harder (and often query specific) metric to provide.

To measure fidelity, interior nodes aggregate C^2 for

individual panes as they make their way up each reduce tree. Since each pane is by definition temporally complete, representing data for that portion of the window, this per-pane C^2 simply maintains a count and the IDs of data sources summarized in a particular pane. As panes are merged in the aggregation tree, so too is their C^2 information. The root represents C^2 as a histogram with a bin per pane that counts the nodes that responded for that pane. This allows the root to summarize C^2 as the percent of nodes reporting (unique nodes responding divided by total nodes) and the percent window computed (non-empty panes divided by total panes per window).

3.2 Using C^2: applications

This section examines how applications use C^2 to bound result quality and to understand imperfect results. Users specify minimum fidelity requirements by annotating queries with a target fidelity that constrains results along particular spatial and temporal dimensions. For example, applications may specify C^2 as a minimum area A, giving the system a large degree of freedom to meet fidelity requirements, as any set of panes will do. Or applications may specify C^2 as percentages of temporal and spatial completeness: $(\%time, \%space)$. For example, one could require panes in the window to be 100% spatially complete (as they are in the left-hand of Figure 5), but relax the requirement for the other axis.

The goal for an application is to set a fidelity bound that allows users to determine result quality from C^2. In particular, they should fix the axes along which result quality is unpredictable. Thus two results may both meet the fidelity bound, but users can ascertain relative result quality by comparing how they did so. To illustrate these concepts, we now describe four general C^2 specifications and their fidelity/latency tradeoffs.

Area (A) with earliest results: This C^2 specification gives the system the most freedom to decrease result latency (or shed load). Without failure or load shedding, iMR will return the first $A\%$ panes from each log server for the result window. These results will correctly summarize event frequencies only if events were uniformly distributed across the log servers. This is the case with simple applications, such as Word Count, where an approximate answer could be used to estimate the relative frequency of words. However, if some words (events) are associated with some servers more than other words, the data will be biased.

Area (A) with random sampling: This C^2 specification gives the system less freedom to decrease result latency, but tries to ensure that a partial result correctly reproduces the relative occurrence of events in the result window, no matter how events are distributed across the log servers. Here each iMR node randomly creates panes with a probability in proportion to A. This takes longer to reach the fidelity bound than the first strategy, but will correctly sample the log data. Note applications must check the C^2 score to verify a sufficient sample in the event of pane loss due to node or network failures.

Spatial completeness $(X, 100\%)$: This specification ensures that each pane in the result window contains data from 100% of the nodes in the system. It is useful for applications that must correlate log events on different servers that occur close in time. For example, consider a basic click-stream analysis that allows web sites to characterize user behavior. With load-balanced web and application serving architectures, a user's click events may arrive at any log server. Intuitively this C^2 specification captures a spatial "slice" of the log data, collecting a snapshot of user activity across the servers during a pane.

Temporal completeness $(100\%, Y)$: This specification ensures that Y percent of the nodes in the system respond with 100% of the panes in the result window. It is useful for applications that must correlate log events on the same server across time. For example, if in the click-stream analysis, individual users had been assigned/pinned to particular servers, this would be the C^2 to employ.

3.3 Result eviction: trading fidelity for availability

iMR allows users to specify latency and fidelity bounds on continuous MapReduce queries. Here we describe the policies that determine when the root evicts results. The root has final authority to evict a window and it uses the window's completeness, C^2, and latency to determine eviction. Thus a latency-only eviction policy may return incomplete results to meet the deadline, while a fidelity-only policy will evict when the results meet the quality requirement.

Latency eviction: A query's latency bound determines the maximum amount of time the system spends computing each successive window. If the timeout period expires, the operator evicts the window regardless of C^2. Before the timeout, the root may evict early under three conditions: if the window is complete before the timeout, if it meets the optional fidelity bound C^2, or if the system can deduce that further delays will not improve fidelity. Like the root, interior nodes also evict based on the user's latency deadline, but may do so before the deadline to ensure adequate time to travel to the root [23].

Fidelity eviction: The fidelity eviction policy delivers results based on a minimum window fidelity at the root. As panes arrive from nodes in the network, the root updates C^2 for the current window. When the fidelity reaches the bound the root merges the existing panes in

the window and outputs the answer.

Failure eviction: Just as the system evicts results that are 100% complete, the system may also evict results if additional wait time can not improve fidelity. This occurs when nodes are heavily loaded or become disconnected or fail. iMR employs *boundary* panes (where traditional stream processors use boundary tuples [26]) to distinguish between failed nodes and stalled or empty data streams[2]. Nodes periodically issue boundary panes to their parents when panes have been skipped because of a lack of data or load shedding (Section 3.4).

Boundary panes allow the root to distinguish between missing data that may arrive later and missing data that will never arrive. iMR maintains boundary information on a per-pane basis using two counters. The first counter is the C^2 completeness count; the number of successful pane merges. Even if a child has no local data for a pane, its parent in the aggregation tree may increase the completeness count for this pane. However, children may skip panes either because they re-started later in the stream (Section 4.4.2) or because they canceled processing to shed load (Section 3.4). In these cases, the parent node increases an *incompleteness* counter indicating the number of nodes that will never contribute to this pane.

Both interior nodes and the root use these counts to evict panes or entire windows respectively. Interior nodes evict early if the panes are complete or the sum of these two counters is equal to the sum of the children in this sub tree. The root determines whether or not the user's fidelity bound can ever be met. By simply subtracting incompleteness from the total node count (perfect completeness), the root can set an upper bound on C^2 for any particular window. If this estimate of C^2 ever falls below the user's target, the root evicts the window.

Note that the use of fidelity and latency bounds presumes that the user either received a usable result or cannot wait longer for it to improve. Thus, unlike other approaches, such as tentative tuples [8] or re-running the reduction phase [10], iMR does not, by default, update evicted results. iMR only supports this mode for debugging or determining a proper latency bound, as it can be expensive, forcing the system to repeatedly re-process (re-reduce) a window on late updates.

3.4 Load cancellation and shedding

When the root evicts incomplete windows, nodes in the aggregation tree may still be processing panes for that window. This may be due to panes with inordinate amounts of data or servers that are heavily loaded (have little time for log processing). Thus they are computing and merging panes that, once they arrive at the root, will no longer be used. This section discusses mechanisms that cancel or shed the work of creating and merging panes in the aggregation tree. Note that iMR assumes that mechanisms already exist to apportion server resources between the server's normal duties and iMR jobs. For instance, iMR may run in a separate virtual machine, letting the VM scheduler allocate resources between log processing and VMs running site services. Here our goal is to ensure that iMR nodes use the resources they are given effectively.

iMR's load cancellation policies try to ensure that internal nodes do not waste cycles creating or merging panes that will never be used. When the root evicts a window because it has met the minimum C^2 fidelity requirement, there is almost surely outstanding work in the network. Thus, once the root determines that it will no longer use a pane, it relays that pane's index down the aggregation tree. This informs the other nodes that they may safely stop processing (creating/merging) the pane.

In contrast, iMR's load shedding strategy works to prevent wasted effort when individual nodes are heavily loaded. Here nodes observe their local processing rates for creating a pane from local log records. If the expected time to completion exceeds the user's latency bound, it will cancel processing for that pane. It will then estimate the next processing deadline that it can meet and skip the intervening panes (and send boundary panes in their place).

Internal nodes also spend cycles (and memory) merging panes from children in the aggregation tree. Here interior nodes either choose to proceed with pane merging or, in the event that it violates the user's latency bound, "fast forward" the pane to its immediate parent. As we shall see in Section 5, these policies can improve result fidelity in the presence of straggler nodes.

4 Prototype

Our implementation of in-situ MapReduce builds upon Mortar, a distributed stream processing system [22]. We significantly extended Mortar's core functionality to support the semantics of iMR and the MapReduce programming model along four axes:

- Implement the iMR MapReduce API using generic map and reduce Mortar operators.
- Pane-based continuous processing with flow control.
- Load shedding/cancellation and pane/window eviction policies.
- Fault-tolerance mechanisms, including operator restart and adaptive tuple routing schemes.

[2] In reality, all panes contain boundary meta data, but nodes may issue panes that are otherwise empty except for this meta data.

4.1 Building an in-situ MapReduce query

Mortar computes continuous in-network aggregates across federated systems with thousands of nodes. This is a natural fit for the map, combine, and reduce functions since they are either local per-tuple transforms (map) or often in-network aggregates. A Mortar query consists of a single operator, or aggregate function, which Mortar replicates across nodes that produce the raw data streams. These in-situ operators give iMR the opportunity to actively filter and reduce intermediate data before it is sent across the network. Each query is defined by its operator type and produces a single, continuous output data stream. Operators push, as opposed to the pull-based method used in Hadoop, tuples across the network to other operators of the same type.

Mortar supports two query types: local and in-network queries. A local query processes data streams independently at each node. In contrast, in-network queries use a tree of operators to aggregate data across nodes. Either query type may subscribe to a local, raw data source such as a log file, or to the output of an existing query. Users compose these query types to accomplish more sophisticated tasks, such as MapReduce jobs.

Figure 6 illustrates an iMR job that consists of a local query for map operators and an in-network query for reduce operators. Map operators run on the log servers and partition their output among co-located reduce operators (here there are two partitions, hence two reduce trees). The reduce operator does most of the heavy lifting, grouping key-value pairs issued by the map operators before calling the user's combine, uncombine, and reduce functions. Unlike traditional MapReduce architectures, where the number of reducers is fixed during execution, iMR may dynamically add (or subtract) reducers during processing.

4.2 Map and reduce operators

Like other stream processors, Mortar uses processing windows to bound computation and provides a simple API to facilitate programming continuous operators. We implemented generic map and reduce operators using this API to call user-defined MapReduce functions at the appropriate time and properly group the key-value pairs. We modified operator internals so that they operate on panes as described in Section 2.3. Operators take as input either raw records from a local log or they receive panes from upstream operators in the aggregation tree. Internally, iMR represents panes as (possibly sorted) hash maps to facilitate key-value grouping.

In iMR operators have two main tasks: pane creation, creating an initial pane from a local data source, and pane merging, combining panes from children in an aggregation tree. Pane creation operates on a record-by-record basis, adding new records into the current pane. In contrast, pane merging combines locally produced panes with those arriving from the network. Because of differences in processing time and network congestion, operators maintain a sequence of panes that the system is actively merging (they have not yet been evicted). We call this the active pane list or APL.

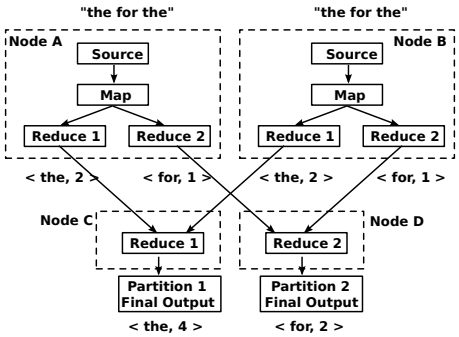


Figure 6: Each iMR job consists of a Mortar query for the map and a query for the reduce. Here there are two MapReduce partitions ($r = 2$), which result in two aggregation trees. A word count example illustrates partitioning map output across multiple reduce operators.

To adapt Mortar for MapReduce processing, we introduce immutable timestamps into the system. Mortar assumes logically independent operators that timestamp output tuples at the moment of creation. In contrast, iMR defines processing windows with respect to the original timestamps on the input logs, not with respect to the time at which an operator was able to evict a pane. iMR assigns a timestamp to each data record when it first enters the system (using a pre-existing timestamp from the log entry, or the current real time). This timestamp remains with the data as it travels through successive queries. Thus networking or processing delays do not alter the window in which the data belongs.

4.2.1 The map operator

The simplicity of mapping allows a streamlined map operator. The operator calls the user's map function for each arriving tuple, which may contain one or more log entries[3]. For each tuple, the map operator emits zero or more key-value pairs. We optimized the map operator by permanently assigning it a tuple window with a range and slide equal to one. This allowed us to remove window-related buffering and directly issue tuples containing key-value pairs to subscribed operators. Finally,

[3]Like Hadoop, iMR includes handlers that interpret log records.

the map operator partitions key-value pairs across subscribed reduce operators.

4.2.2 The reduce operator

The reduce operator handles the in-network functionality of iMR including the grouping, combining, sorting and reducing of key-value pairs. The operators maintain a hash map for each pane in the active pane list. Here we describe how the reduce operator creates and merges panes.

After a reduce operator subscribes to a local map operator it begins to receive tuples (containing key-value {k,v} pairs). The reducer operator first checks the logical timestamp of each {k,v} pair. If it belongs to the current pane, the system inserts the pair into the hash table and calls the combiner (if defined). When a {k,v} pair arrives with a timestamp for the next pane, the system inserts the prior pane into the active-pane list (APL). The operator may skip panes for which there is no local data. In that case, the operator inserts boundary panes into the APL with completeness counts of one.

Load shedding occurs during pane creation. As tuples arrive, the operator maintains an estimate of when the pane will complete. The operator periodically updates this estimate, maintained as an Exponentially Weighted Moving Average (EWMA) biased towards recent observations (α = 0.8), and determines whether the user's latency deadline will be met. For accuracy, the operator processes 30% of the pane before the first estimate update. For responsiveness, the operator periodically updates and checks the estimate (every two seconds). For each skipped pane the operator issues a boundary pane with an incompleteness count of one.

The APL merges locally produced panes with panes from other reduce operators in the aggregation tree. The reduce operator calls the user's combiner for any group with new keys in the pane's hash map. The operator periodically inspects the APL to determine whether it should evict a pane (based on the policies in Section 3.3). Reduce operators on internal or leaf nodes forward the pane downstream on eviction.

If the operator is at the tree's root, it has the additional responsibility of determining when to evict the entire window. The operator checks eviction policies on periodic timeouts (the user's latency requirement) or when a new pane arrives (possibly meeting the fidelity bound). At that point, the operator may produce the final result either by using the optional uncombine function or by simply combining the constituent panes (strategies discussed in Section 2.3). After this combining step, the operator calls the user-defined reduce function for each key in the window's hash map.

4.3 Pane flow control

Recall that the goal of load shedding in iMR isn't to use less resources, but to use the given resources effectively. Given a large log file, load shedding changes the work done, not its processing rate. Thus it is still possible for some nodes to produce panes faster than others, either because they have less data per pane or more cycles available. In these cases, the local active pane list (APL) could grow in an unbounded fashion, consuming server memory and impacting its client-facing services.

We control the amount of memory used by the APL by employing a window-oriented flow control scheme. Each operator monitors the memory used (by the JVM in our implementation) and issues a pause indicator when it reaches a user-defined limit. The indicator contains the logical index of the youngest pane in the operator's APL. Internally, pane creation waits until the indicator is greater than the current index or the indicator is removed. Pause indicators are also propagated top-down in the aggregation tree, ensuring that operators send evicted panes upward only when the indicator is greater than the evicted indices or it is not present.

4.4 MapReduce with gap recovery

While load shedding and pane eviction policies improve availability during processing and network delays, nodes may fail completely, losing their data and current queries. While traditional MapReduce designs, such as Hadoop, can restart map or reduce tasks on any node in the cluster, iMR does not assume a shared filesystem. Instead, iMR provides *gap recovery* [19], meaning that the system may drop tuples (i.e., panes) in the event of node failures.

4.4.1 Multi-tree aggregation

Mortar avoids failed network elements and nodes by routing data up multiple trees. Nodes route data up a single tree until the node stops receiving heart beats from its parent. If a parent becomes unreachable, it chooses another tree (i.e., another parent) to route tuples to. For this work, we use a single tree; this simplifies our implementation of failure eviction policies because internal nodes know the maximum possible completeness of panes arriving from their children.

Mortar employs new tuple routing rules to retain a degree of failure resilience. If a parent becomes unreachable, the child forwards data directly to the root. This policy allows data to bypass failed nodes at the expense of fewer aggregation opportunities. Mortar also designs its trees by clustering network coordinates [11], and we use the same mechanism in our experiments. We leave more advanced routing and tree-building schemes as future work.

4.4.2 Operator re-install

iMR guarantees that queries (operators) will be installed and removed on nodes in an eventually consistent manner. Mortar provides a reconciliation algorithm to ensure that nodes eventually install (or un-install) query operators. Thus, when nodes recover from a failure, they will re-install their current set of operators. While we lose the data in the operator's APL at the time of failure, we need to re-start processing at an appropriate point to avoid duplicate data. To do so, operators, during pane creation, maintain a simple on-disk write-ahead log to indicate the next safe point in the log to begin processing on re-start. For many queries the cost of writing to this log is small relative to pane computation, and we simply point to the next pane.

5 Evaluation

Our evaluation explores both the baseline performance of our prototype and the ability of our system to deliver results in the event of delays or failures. Unless noted otherwise, we evaluated iMR on a 40 node cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24 GB of memory, and 16 HP 507750-B21 500GB 7,200 RPM 2.5" SATA drives. Each server has two HP P410 drive controllers, as well as a Myricom 10 Gbps network interface. The network interconnect we use is a 52-port Cisco Nexus 5020 datacenter switch. The servers run Linux 2.6.35, and our implementation of iMR is written in Java. iMR experiments use star aggregation topologies.

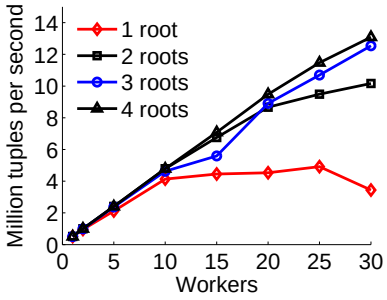
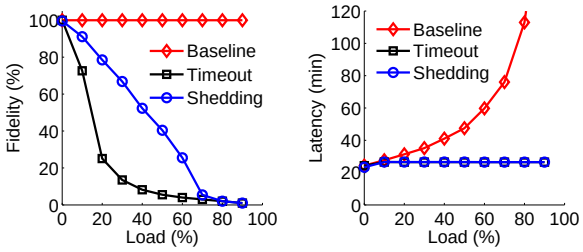


Figure 7: Scaling iMR as the number of workers (and processing nodes) increases.

5.1 Scaling

We first establish the scale-out properties of our processing architecture. The purpose of these experiments is to verify the ability of the system to scale as we increase both the number of mappers and the number of reducer partitions. Here we use synthetic input data and a reducer that implements a word count function. The query uses a tumbling window where the range is equal to the slide; in this case the window range is 150 million input records, approximately 1GB of input data. We allow the job to run for five minutes and take the average throughput. Unlike Hadoop, the iMR job is configured to read the log from local disk.

Figure 7 plots the records per second throughput of iMR as we increase the total cluster capacity. Each line represents a different configuration that increases the reducer and physical node count by one. Here three reducers provide sufficient processing to handle the 30 map tasks. We see that, as long as the reducer is not the bottleneck, adding additional nodes increases throughput linearly. Similarly, reducers can also add a linear increase in throughput.



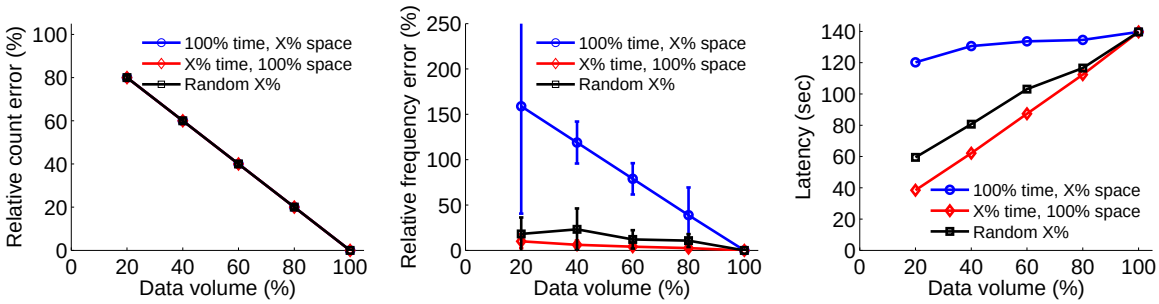
(a) Fidelity (b) Latency

Figure 8: Impact of load shedding on fidelity and latency for a word count job under maximum latency requirement and varying worker load.

5.2 Load shedding

These iMR experiments evaluate the ability of load shedding to improve result fidelity under limited CPU resources. We execute a word count MapReduce query on a single node; this node installs a single map and reduce operator. We vary the CPU load by running a separate CPU burn application. The query specifies a tumbling window ($R = S$) that contains 20 million records and we configure the system to use 20 panes per window. We execute the query until it delivers 10 results and report the average latency (Figure 8(a)) and fidelity (Figure 8(b)) as we increase CPU load.

The baseline query has no latency requirement and always delivers results with 100% fidelity. The timeout query has a latency requirement equal to the observed baseline window latency, which is 160 seconds. Though results meet the latency requirement, quality degrades as the load increases. Without load shedding the worker attempts to process all panes, even if very few can be delivered in time. In contrast, load shedding allows the



(a) Count error (b) Frequency error. (c) Result latency.

Figure 9: The performance of a count statistic on data skewed across the log server pool. Enforcing either random pane selection or spatial completeness allows the system to approximate count frequencies and lower result latency.

worker to use the available CPU intelligently, processing only the panes that can be delivered on time and increasing average fidelity substantially.

5.3 Failure eviction

Here we show how failure eviction can deliver results early if nodes fail. We execute a word count MapReduce query on 10 workers. The query uses a tumbling window with 2 million records, 2 panes, and a 30 second latency requirement. After starting the query, we emulate transient failures by stopping an increasing number of workers. The experiment finishes when the query delivers 20 results.

In Figure 10, we report application goodput as the number of panes delivered to the user per time. Note that this metric is not a direct measure of how fast workers can process raw data. Instead it reflects the ability of the system to detect failures and deliver panes to the user early. The higher the metric, the less the user waits to get the same number of panes. Without failure eviction the root times out (30 seconds) before it delivers incomplete results. With failure eviction, the root can deliver results before the timeout, improving goodput by 57-64%.

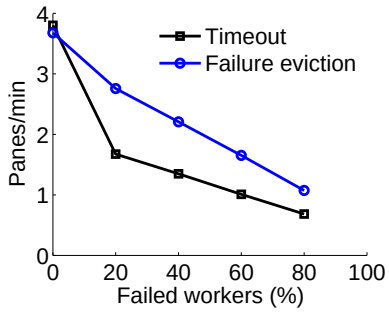


Figure 10: Application goodput as the percentage of failed workers increases. Failure eviction delivers panes earlier, improving goodput by up to 64%.

5.4 Using C^2

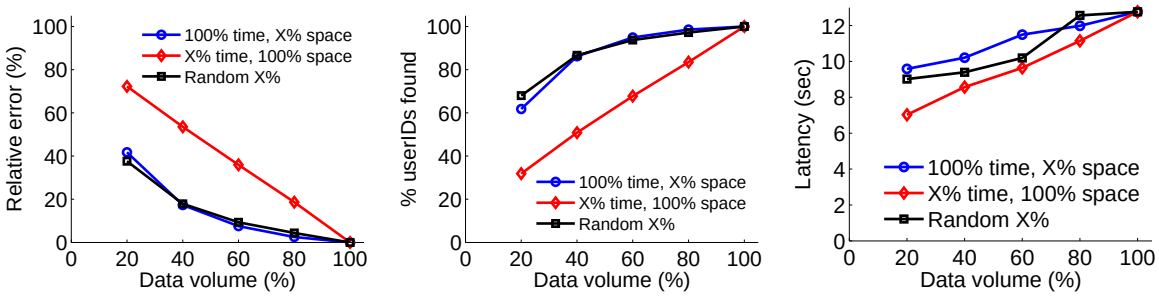
This section explores how we use the C^2 framework for three different application scenarios: word count with non-uniformly distributed keys, click-stream analysis, and an HDFS anomaly detector. These experiments used a 30-node cluster of Dual Intel Xeon 2.4GHz machines with 4GB of RAM connected by gigabit Ethernet.

5.4.1 Word Count

Our first experiment performed a word count query across synthetic data placed on ten log servers in our local cluster. This configuration allows us to explore the impact of different fidelity bounds on absolute count estimations and relative word frequency. We distribute the words in the synthetic data across the log servers in a skewed fashion, where some words are more likely to be on some servers than others. In these experiments the window range (and slide) is 100MB, the pane size is 10 MB, and there is no latency bound.

Here we explore three different C^2 settings: temporal completeness, spatial completeness, and area with random pane selection. Figure 9 shows the relative error in reported count, the relative error of the word frequency (with std. dev.), and the result latency as we increase the data fidelity. As expected, the count error (Figure 9(a)) improves linearly as we force the system to include more data in each window (data volume).

However, because the data are not uniformly distributed, the frequency error (Figure 9(b)) is large for the temporal completeness C^2 specification, $(100\%, Y)$. Note in this experiment we achieve varying levels of temporal completeness by randomly selecting specific nodes to fail to report for an entire window. By removing data from a source completely, some keys may completely lose their representation and the remaining key's fre-



(a) Average error per userID (b) Percentage userID's found. (c) Result latency.

Figure 11: Estimating user session count using iMR and different C^2 policies. Random pane selection and temporal completeness provide significantly higher data fidelity than enforcing spatial completeness.

quencies shift. Both random pane selection and spatially complete results do much better, since they effectively sample from the entire server pool.

Finally, these three policies differ substantially in the latency of the results they deliver. Figure 9(c) plots the result latency for each C^2 specification. Clearly, providing temporal completeness requires each node to finish processing the entire window before returning a result. In contrast, by asking for spatial completeness, the root can return as soon as the first x% of the panes complete, allowing the best latency.

5.4.2 Click-stream analysis

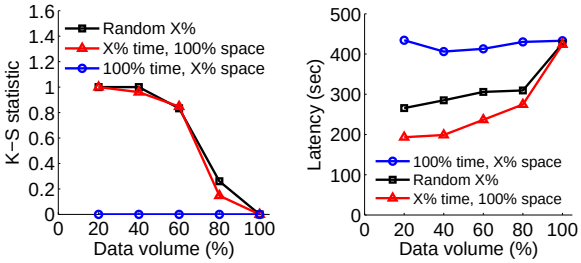
Here we develop a simple click-stream analysis. This analysis takes as input a log of click records that contain userID and timestamp fields. We developed a reduce function to calculate three different click analysis metrics: the number of user sessions, the average session duration, and the average number of clicks per session. We use our different C^2 specifications and study the relative error each provides.

These experiments use 24 hours of publicly available server logs from the 1998 World Cup [1] as input. We partition this data (4.5GB in total) across ten of our servers, preserving the characteristic that clicks from a single user are often served by different nodes in the trace. The window (and slide) of the MapReduce job is set to two hours and we set the pane size to be 6 minutes (20 panes per window). We run each query for the entire data set (12 windows).

Figure 11 shows how the number of sessions per user changes as we accept different levels of data fidelity. Surprisingly, requiring data from all nodes for each pane, $(X, 100\%)$, leads to large relative errors (per user). This is primarily because userIDs are not uniformly distributed across time and enforcing spatial completeness does not give a decent sample. However, randomly sampling at each log server lowers relative error to 20% (per user), even when computing across less than 50% of the window's data.

Figure 11(b) shows that those policies also recover a large fraction of the total userID space even when they sample a relatively small total fraction of data. Thus for this application, the best C^2 specification is random pane selection, as it not only provides the best results but also allows the system to lower result latency as well (Figure 11(c)).

5.4.3 HDFS log analysis



(a) KS-test (b) Latency

Figure 12: (a) Results from the Kolmogorov-Smirnov test illustrate the impact of reduced data fidelity on the histograms reported for each HDFS server. (b) For HDFS anomaly detection, random and spatial completeness C^2 improve latency by at least 30%.

Our last application analyzes logs from the Hadoop distributed file system (HDFS) to determine faulty storage nodes. The iMR MapReduce job first filters the local HDFS log, finding all unique block write events. The reduce function then computes a histogram of the block write service times. This collection of histograms, one per HDFS server, is then analyzed to determine anomalies in the cluster [27].

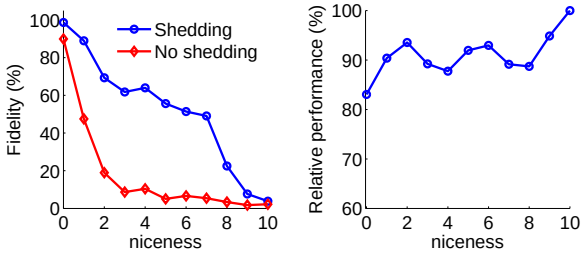
We generated 48 hours of HDFS logs by running the

GridMix Hadoop workload generator [3] on our 30-node cluster. Each node's log is approximately 2.5 GB, yielding appr. 75 GB in total. This analysis compares the quality of the histograms produced under different C^2 specifications to the histogram produced with no loss. The query has a window range (and slide) of 48 hours and uses 1 hour panes.

We use the Kolmogorov-Smirnov test to compare the per-server histograms with perfect and incomplete data. Figure 12(a) shows the percentage of histograms that when using incomplete data represent a markedly different distribution (reject the null hypothesis). Here the $(100\%, Y)$ policy generates perfect data, since, if a node reports, all data is included. The other C^2 strategies result in a majority of the histograms failing the null hypothesis when using less than 80% of the data.

However, since those strategies can lower result latency significantly at that data volume (about 30% in Figure 12(b)), users must decide whether that is an acceptable tradeoff. Going forward we intend to look at how this ultimately impacts the ability to find failing HDFS nodes.

5.5 In-situ performance



(a) Fidelity (b) Hadoop performance

Figure 13: Fidelity and Hadoop performance as a function of the iMR process niceness. Hadoop is always given the highest priority, nice $= 0$.

We designed iMR to effectively process log data "on location." This experiment illustrates the ability of the iMR architecture to produce useful results when run side-by-side with a real application. Specifically, our 10-node cluster will execute Hadoop and iMR simultaneously. Here, Hadoop executes a workload generated by the GridMix generator and iMR executes a word count query with a window of 2 million records, 20 panes per window, and a 60 second timeout. We vary the CPU allocated to iMR by changing the priority (niceness) assigned to the iMR process by the kernel scheduler and report the average result fidelity. We also report the relative change in the Hadoop performance, in terms of jobs completed per time. Each data point is the average of five runs.

Figure 13(a) shows that without load shedding, result fidelity falls almost linearly as the iMR process' priority decreases. In contrast, load shedding greatly improves fidelity until there is insufficient CPU remaining to process any pane by the deadline (nice $= 9$). Looking at Hadoop performance in Figure 13(b), we see that the cost for giving them equal priorities is a decrease in job throughput of 17%. Even when using nice, a relatively coarse-grain knob for resource allocation, to assign a lower priority to log processing, Hadoop can improve job throughput ($< 10\%$ penalty) and iMR can still deliver useful results.

6 Related work

"Online" bulk processing: iMR focuses on the challenges of migrating initial log analytics to the data sources. A different (and complementary) approach has been to optimize traditional MapReduce architectures for log processing themselves. For instance, the Hadoop Online Prototype (HOP) [10] can run continuously, but requires custom reduce functions to manage their own state for incremental computation and framing incoming data into meaningful units (windows). iMR's design avoids this requirement by explicitly supporting sliding window-based computation (Section 2.1), allowing existing reduce functions to run continuously without modification.

Like iMR, HOP also allows incomplete results, producing "snapshots" of reduce output, where the reduce phase executes on the map output that has accumulated thus far. HOP describes incomplete results with a "progress" metric that (self admittedly) is often too coarse to be useful. In contrast, iMR's C^2 framework (Section 3) not only provides both spatial and temporal information about the result, but may be used to trade particular aspects of data fidelity for decreased processing time.

Dremel [24] is another system that, like iMR, aims to provide fast analysis on large-scale data. While iMR targets continuous raw log data, Dremel focuses on static nested data, like web documents. It employs an efficient columnar storage format that is benefitial when a fraction of the fields of the nested data must be accessed. Like HOP, Dremel uses a coarse progress metric for describing early, partial results.

Log collection systems: A system closely related to iMR is Flume [2], a distributed log collection system that places *agents* in-situ on servers to relay log data to a tier of collectors. While a user's "flows" (i.e., queries) may transform or filter individual events, iMR provides a more powerful data processing model with grouping, reduction, and windowing. While Flume supports best-

effort operation, users remain in the dark about result quality or latency. However, Flume does provide higher reliability modes, recovering events from a write-ahead log to prevent data loss. While not discussed here, iMR could employ similar *upstream backup* [19] techniques to better support queries that specify fidelity bounds.

Load shedding in data stream processors: iMR's load shedding (Section 3.4) and result eviction policies (Section 3.3) build upon the various load shedding techniques explored in stream processing [9, 28, 29]. For instance, iMR's latency and fidelity bounds are related to the QoS metrics found in the Aurora stream processor [9]. Aurora allows users to provide "graphs" which separately map increased delay and percent tuples lost with decreasing output quality (QoS). iMR takes a different approach, allowing users to specify latency and fidelity bounds above which they'd be satisfied. Additionally, iMR leverages the temporal and spatial nature of log data to provide users more control than percent tuples lost.

Many of these load shedding mechanisms insert tuple dropping operators into query plans and coordinate drop probabilities, typically via a centralized controller, to maintain result quality under high-load conditions. In contrast, our load shedding policies act locally at each operator, shedding sub-windows (panes) as they are created or merged. These "pane drop" policies are more closely related to the probabilistic "window drop" operators proposed by Tatbul, et al. [29] for aggregate operators. In contrast, iMR's operators may drop panes both deterministically or probabilistically depending on the C^2 fidelity bound.

Distributed aggregation: Aggregation trees have been explored in sensor networks [23], monitoring wired networks [31], and distributed data stream processing [18, 22]. More recent work explored a variety of strategies for distributed GroupBy aggregation required in MapReduce-style processing [32]. Our use of sub-windows (panes) is most closely related to their *Accumulator-PartialHash* strategy, since we accumulate (through combining) key-value pairs into each sub-window. While they evicted the sub window based on its storage size (experiencing a hash collision), iMR uses fixed-sized panes.

7 Conclusion

This work explores moving initial log analysis steps out of dedicated clusters and onto the data sources themselves. By leveraging continuous in-situ processing, iMR can efficiently extract and transform data, improving system scalability and reducing analysis times. A key challenge is to provide a characterization of result fidelity that allows users to interpret results in the face of incomplete data. For a handful of applications, we illustrated how the C^2 framework allows users to explicitly trade specific aspects of data fidelity in the event failures lose data or the system cannot meet latency requirements. Future work will consider how the system can assist in setting appropriate C^2 fidelity bounds, and whether similar techniques could be applied in dedicated processing cluster environments.

Acknowledgements

We'd like to thank Geoff Voelker and our shepherd, Leendert van Doorn, for their helpful feedback. This work was supported in part by the National Science Foundation through grant CCF-1048296.

References

[1] 1998 World Cup Web Server Logs. http://ita.ee.lbl.gov/html/traces.html.

[2] Flume: Open source log collection system. http://github.com/cloudera/flume.

[3] The GridMix Hadoop Workload Generator. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[4] Windows Azure and Facebook teams. Personal communications, August 2008.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2005.

[6] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI'10*, April 2010.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems*, March 2002.

[8] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD'05*, June 2005.

[9] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB'02*, September 2002.

[10] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *NSDI'10*, San Jose, CA, April 2010.

[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM'04*, August 2004.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, San Francisco, CA, December 2004.

[13] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, alex Russakovsky, and N. Thomre. Continuous analytics: Rethinking query processing in a network-effect world. In *Proc. of CIDR*, January 2009.

[14] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[16] B. He, M. Yang, zhenyu Guo, R. Chen, W. Lin, B. Su, and L. Zhou. Batched stream processing: A case for data intensive distributed computing. In *ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[17] T. Hoff. How Rackspace Now Uses MapReduce and Hadoop To Query Terabytes of Data, Jan 2008. http://highscalability.com/how-rackspace-now-uses-mapreduce-and-hadoop-query-terabytes-data.

[18] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of VLDB*, September 2003.

[19] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of ICDE*, April 2005.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys'07*, March 2007.

[21] J. Li, D. Maier, K. Tufte, V. Papdimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1), March 2005.

[22] D. Logothetis and K. Yocum. Wide-scale data stream management. In *USENIX Annual Technical Conf.*, June 2008.

[23] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI'02*, December 2002.

[24] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel : Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3, September 2010.

[25] R. N. Murty and M. Welsh. Towards a dependable architecture for Internet-scale sensing. In *Second HotDep06*, November 2006.

[26] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of PODS 2004*, June 2004.

[27] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: analyzing logs as state machines. In *1st USENIX Workshop on Analysis of System Logs*, page 6, December 2008.

[28] N. Tatbul, U. Cetintemel, and S. Zdonik. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *VLDB'07*, August 2007.

[29] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB'06*, August 2006.

[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques)*, Dec 2008.

[31] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM'04*, September 2004.

[32] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP'09*, October 2009.