◆ Towards the Optimization of a Parallel Streaming Engine for Telco Applications

Bart Theeten, Ivan Bedini, Peter Cogan, Alessandra Sala, and Tommaso Cucinotta

Parallel and distributed computing is becoming essential to process in real time the increasingly massive volume of data collected by telecommunications companies. Existing computational paradigms such as MapReduce (and its popular open-source implementation Hadoop) provide a scalable, fault tolerant mechanism for large scale batch computations. However, many applications in the telco ecosystem require a real time, incremental streaming approach to process data in real time and enable proactive care. Storm is a scalable, fault tolerant framework for the analysis of real time streaming data. In this paper we provide a motivation for the use of real time streaming analytics in the telco ecosystem. We perform an experimental investigation into the performance of Storm, focusing in particular on the impact of parameter configuration. This investigation reveals that optimal parameter choice is highly non-trivial and we use this as motivation to create a parameter configuration engine. As first steps towards the creation of this engine we provide a deep analysis of the inner workings of Storm and provide a set of models describing data flow cost, central processing unit (CPU) cost, and system management cost. © 2014 Alcatel-Lucent.

Introduction

Telecom companies are increasingly seeing the need for big data platforms to handle the vast quantity of data generated within their networks. Such data includes performance logs, call data records, customer experience data, and fault reports. MapReduce has become a popular approach, both in industry and academia, for the batch analysis of large quantities of data and is a strong candidate for many telecom analytic applications. This is due to its scalability, fault tolerance, and relatively simple distributed programming paradigm which require only a small number of functions for user implementation. However, the batch processing nature of MapReduce, which requires that the full dataset is available at the start of the analysis, may make it unsuitable for certain applications within the telco ecosystem. For instance if a backend server is producing a continuous stream of log data, these logs may contain early indications of network issues which the telecom providers must address as quickly as possible to ensure quality of service to subscribers. Under the MapReduce paradigm, data would be aggregated over some time period τ, and provided to MapReduce for batch analysis. The

Bell Labs Technical Journal 18(4), 181–197 (2014) © 2014 Alcatel-Lucent. Published by Wiley Periodicals, Inc.
Published online in Wiley Online Library (wileyonlinelibrary.com) • DOI: 10.1002/bltj.21652

computational cluster upon which the analysis is performed should be scaled such that the time, T, taken to complete the analysis should be smaller than τ, such that the analysis is complete before the next data aggregation arrives. As a result, all results will be at least T old and at most $(T + \tau)$ old.

Streaming analysis is seen as an emerging alternative to the batch computation approach used by MapReduce. Under this paradigm, data are considered as streams of tuples which are transformed and merged from different sources to create a final stream of analyzed results. In this scenario, the age of the analyzed results is just given by the system traversal time (i.e., latency L), as long as the computational cluster is horizontally scaled such that the throughput rate is greater than the data arrival rate. Note that we would expect $L << T$.

The most important consideration when determining whether it is appropriate to use a batch or streaming analysis is the specific algorithm that must be deployed. For example, an algorithm such as PageRank* requires multiple passes over the data which in turn requires that the entire dataset can be stored and is available. In such a case, a batch analysis is appropriate. In contrast, if data requires single pass analysis (such as in some clustering techniques [3]), and/or the data cannot be stored for multiple passes, then streaming analysis is the most appropriate choice. Furthermore, a streaming analysis can take advantage of incremental algorithms for updating results in real time for computations such as the mean, maximum, or minimum of some parameter. This approach facilitates live business intelligence applications, which is extremely useful in the telecom ecosystem. We provide more detailed examples in the "Use Cases" section below.

Hadoop* and Storm*, respectively, are open source implementations of MapReduce and streaming analytics. Both of these frameworks provide horizontally scalable, fault tolerant platforms for distributed computation. Hadoop has enjoyed considerable success as a mainstream distributed analytics platform, with many applications deployed in industrial production systems. Storm is a relatively new framework for distributed computation of real time streaming data.

Panel 1. Abbreviations, Acronyms, and Terms

3GPP—3rd Generation Partnership Project
BW—Bandwidth
CDR—Call data record
CPU—Central processing unit
HDFS—Hadoop Distributed File System
JVM—Java virtual machine
M3—Main-Memory MapReduce
MMS—Multimedia messaging service
OAM—Operations, administration, and maintenance
PM—Performance management
SMS—Short message service
SPADE—Stream Processing Application Declarative Engine
XML—Extensible Markup Language

In this paper we present motivations and use cases for streaming analytics within the telecom ecosystem. We perform a series of investigative experiments to better understand the impact of parameter configuration upon Storm performance. In finding that the performance is highly dependent on this parameter tuning, and that a priori selection of optimal parameters is non-trivial, we embark upon an effort to create an automatic engine for the selection of optimal parameters. We present the first steps toward creating this engine by providing a deep description of the inner workings of Storm and by providing models which describe the data flow cost, central processing unit (CPU) cost, and system management costs.

Use Cases

While data is growing at a speed never before seen, today's consumers are increasingly demanding not only "always-on" connectivity and access, but better service quality and overall experience. Thus, operators look to real time analytics as an important enabler to speed-up the creation, delivery, and monetization of service bundles and to provide a unique network experience for their customers. In this context, the creation of new responsive and dynamic scalable solutions for data analytics is becoming essential.

In this section we describe two broad use cases we have considered to build concrete scenarios for the real time big data analysis relevant to telecommunication companies. These include real time monitoring for smallcell performance management, and call data record analysis, where the prospect of capturing data in real time in concert with horizontal scalability are essential for proactive network management and traffic monitoring.

Smallcell Performance Management

Smallcells [5] were designed for use in a home, in a small business, or for boosting the network signal in busy metropolitan areas to improve localized cellular service and offload bandwidth usage from macrocells (i.e., traditional cell towers). During operation, smallcells (as well as most other network elements) produce many low-level performance metrics (e.g., number of successful handovers or number of call initiation attempts) across a range of performance categories (e.g., packet data performance or handover performance). In most currently deployed architectures, this set of data is periodically captured and stored as Extensible Markup Language (XML) following the 3rd Generation Partnership Project (3GPP) specification, either temporarily on the smallcell or in a network management application. This data is usually batch-analyzed to monitor network performance characteristics, prediction of peak loads, and prediction of service interruptions.

As smallcells are deployed in rapidly increasing numbers, from 2.5 million in 2012 to a predicted 59 million in 2015 (an increase of 2500 percent) with further increases projected for the years following, the management of large amounts of operational data generated by the smallcells, and an appropriate rapid response to analytic results, is becoming a key challenge. This data increase represents growth from 50 GB/day to 12 TB/day. These responses are important to assure the stability of the smallcell network, and they offer promising opportunities for dynamic configuration management of the network. Current architectures and infrastructures based on relational databases do not effectively scale to the large amounts of data being generated, while emerging big data technologies provide the potential to both support this large amount of data and facilitate insightful network analytics in near real time.

In our research work we envisage the design and realization of a cost-effective cloud architecture able to support these new real time analytic requirements. In the context of smallcells, this will not be to increase the processing capacity of small cells themselves, but to centralize the processing of performance management data from the entire network in real time.

CDR Management

Telecom exchanges produce call data records (CDRs) when subscribers make calls, send short message service/multimedia messaging service (SMS/MMS) messages, and take other actions on the network. These records contain the date and time of the call, the ID of the subscriber, ID of the call recipient, ID of the cell tower to which the handset is connected, as well as account information. This data is used by the telecom provider for a variety of purposes such as billing and diagnostics. The data can also be used to create advanced business intelligence services such as targeted advertisements [4], to better understand user behavior [17], to predict customers' inclination to churn [18, 22], or to recommend new services [20]. However, as the number of subscribers continues to increase, and the frequency with which CDRs are generated increases, analysis of these records becomes burdensome. Indeed, a batch analysis of this data for aggregations greater than two weeks apart is often impractical due to the sheer data size (typically several TB/day). As a result, a streaming analysis which is horizontally scalable and fault tolerant, such as that discussed in this paper, becomes necessary. In [10] the authors use a custom streaming analytics infrastructure to analyze CDRs at a rate of approximately 6 billion CDRs/day (depending on data size, this corresponds to approximately 5 TB/day). Rather, we seek to leverage the open source Storm framework for real time streaming analytics.

Streaming analytics naturally lends itself to incremental computation, rather than iterative computation.

For example, in the case of standard classification algorithms, streaming analytics may not be ideal for the model training phase, where multiple iterations on the data are typically required to achieve convergence. However, once the model has been created, streaming analytics are ideal for applying the model to new data. Similarly, in the case of clustering, once cluster centers have been established, streaming analytics is ideal for clustering of new data into existing centers. For example, suppose an operator has an existing model for the segmentation of users into large and small wallet size. Using streaming analytics, the operator could identify the wallet size of new subscribers in real time as they start to use the network. With this approach, the operator could rapidly create services targeted to the new subscriber.

The Storm Framework

This section briefly describes the Storm framework. For a more detailed description, the reader is directed to [21]. Storm is a scalable, fault tolerant framework which facilitates the processing of streaming data. The programming model involves the creation of a topology which represents the algorithm to be implemented. A topology consists of spouts and bolts. Spouts create one or more streams of tuples which are injected into the topology, while bolts receive one or more streams of tuples and can optionally output one or more streams of tuples (to other bolts). The topology can be modeled as a directed acyclic graph, where nodes correspond to spouts, and bolts and edges represent tuple streams between them. Parallelization is achieved by setting the number of instantiations of each spout and bolt. Typically, a spout or bolt instance is implemented as a Java* thread. Once started, a Storm topology runs continuously on incoming data until it is killed. This is demonstrative of how it is set apart from batch systems such as MapReduce which run on a fixed set of data and then finish. The Storm framework provides mechanisms for automatically distributing processes across the cluster, for directing streams and ensuring fault tolerance.

The utility of big data engines, in an industrial context, is measured not only by flexibility, but also on properties such as fault tolerance, load balancing, and system overhead. Storm is a novel computational engine for processing large scale streams of data. In order to understand the behavior of the Storm framework, we conducted an extensive experimental investigation by running Storm in multiple configurations.

Experimental Investigation of Storm Behavior

This section provides a detailed experimental investigation of the performance characteristics of the Storm system using real industrial datasets. We aim to understand the impact of parameter selection upon the performance of Storm. Furthermore, our investigation sought to determine just how straightforward it is to configure Storm for optimal performance, and to shed light on the precautions required in order to run the system under optimal configurations.

Dataset and Environment Setup

The experiments have been run with a real telco dataset consisting of operations, administration, and maintenance (OAM) performance management (PM) observations of a large femtocell (a specific smallcell) network. The considered dataset is composed of hourly PM logs collected over 15 days for a network of 70K femtocells, totaling approximately 22 million XML files. The scenario of the experiments is such that the system replays the PM data as arriving in a streaming fashion with different arrival rates. This simulation is absolutely coherent and does not provide any significant change to the data values themselves, but allows a good simulation of a real time architecture. These files contain a list of 128 key-value pairs of operational and statistical counters with a mixture of integer and floating-point values. The experiments presented in this paper deploy a simple topology composed of two components (one spout and one bolt) whose task it is to identify those femtocells which require the highest bandwidth. The spout reads input messages from an external queue (this corresponds to the XML data) and produces a stream of tuples. The bolt receives the stream of tuples with a shuffle grouping and emits any changes in the highest-bandwidth-consuming femtocells.

For cluster configuration, each test was run on a cluster comprising five nodes of identical configuration. Additional machines were used to generate load into this cluster and to host the external message queue(s). Each machine is a dual four-core Intel Xeon* 3 GHz 32 bit 16 GB memory, 1 Gb/s network interface. Nodes are interconnected through an Alcatel-Lucent 10Gb OmniSwitch™ 6850 Ethernet switch. Each node runs Linux* version 2.6.32-220.4.1.el6.i686. The following software components were used: Java 1.6 OpenJDK Runtime Environment (IcedTea6 1.10.4), ZeroMQ* 2.1.7, ZooKeeper 3.4.2, and Kestrel 2.3.4 and Storm 0.8.2.

Experiment Configurations

A comprehensive test automation suite was developed which is composed of various shell scripts and Java programs to automatically execute the Storm topologies and collect runtime statistics on system performance. Storm configurations were tested while varying the number of nodes, spouts, bolts, and workers. Each specific configuration was tested 10 times in order to accumulate statistical information. Careful consideration was taken to ensure that each configuration was tested under the same operational conditions.

Each individual test case is executed on a clean cluster, meaning that all processes from a previous run were killed on all cluster nodes before starting the new ones. In addition, all data generated by the previous run is erased from the file system. Each test consists of an initial warm-up phase of one minute (in which we do not collect statistics because the system may not yet be in its computational steady state) followed by a five minute measuring phase during which statistics are being gathered, including:

1. Throughput (μ) which represents the total number of events processed per second.
2. Latency (L) which represents time to process a tuple both within a single bolt process and within the entire system.
3. External and internal queue sizes and their growth rate.
4. Network bandwidth in terms of the number of messages exchanged among workers on different nodes.
5. Various statistics on system management overhead (e.g., communication with ZooKeeper).
6. Approximate memory usage and CPU usage per thread. Each test is run with a different Storm configuration which is determined by the following parameters:
   - *Parallelization factor*. Represents the number of tasks, from {1, 2, 4, 8, 16, 24, 32}, instantiated per spout or a bolt.
   - *Cluster size*. Represents the number of nodes participating in the Storm cluster, i.e., from one to five.
   - *Worker pool size*. Represents the number of workers (JVMs) per node, which host the tasks. In our experiment this number is selected from {1, 2, 4, 8, 16, 24, 32}.
   - *Event injection rate*. Represents the number of events injected per second into the queue, which provides data to the spouts. In our experiments the event injection rate varies among three different rates, 5K, 10K, and 50K tuples per second, to observe how Storm adjusts to different conditions.

Experimental Analysis

In order to optimize system performance while running Storm jobs, there are several variables and parameters which need manual configuration. Configuration is thus a complex task that requires a precise knowledge of the most relevant parameters and how they impact system performance. In particular, we focus on the following metrics: the throughput (μ), the latency (L), and the system resource utilization (CPU, memory, network bandwidth). In the following subsections we present a set of experiments that aim to shed light on how the choice of configuration parameter impacts these metrics.

Parallelization performance. The experiments presented in this section comprise observations of the impact on latency and throughput produced by different configurations in terms of parallelization.

A configuration is expressed as the specification of the system parameters, i.e., number of nodes, spouts, bolts, and workers (hereafter simply $< n, s, b, w >$). For brevity, we only report experiments run

on a single node cluster with an external queue fed with 50K messages per second. However, the same experiments performed on cluster sizes of up to five nodes show similar results, with only a few small deviations attributed to the increased system management costs to run more nodes.

- *Spout parallelization.* Spouts inject tuples into the topology. Increasing the amount of spouts is therefore expected to increase the throughput, as long as the bolts are able to keep up with the higher influx of tuples. At the point where bolts are no longer able to keep up (i.e., fully loaded), it is expected that the throughput will actually decrease because of the additional queue buildup in the system and the larger share of processing power claimed by the many spout instances versus the fixed amount of bolt instances.

  In order to study the impact of the number of spouts on the overall system performance, we fixed all other parameters at 1 while varying the number of spouts at {1, 2, 4, 8, 16, 24, 32}. **Figure 1** illustrates the effect of spout and bolt parallelization on the throughput and latency. Specifically, Figure 1a illustrates the effect upon throughput (in terms of tuples per second) and Figure 1b the effect upon latency (in terms of milliseconds to process a tuple) as the number of spouts is increased. The system throughput can be increased by increasing the number of spouts, however, as the number of spouts continues to increase beyond some threshold, the throughput declines. This can be understood by observing the latency, which exhibits exponential growth. Beyond some threshold (determined by the system hardware), the system is overstressed with many processes and the context-switching among them impairs the system performance.
- *Bolt parallelization.* Bolts process tuples emitted by spouts. Increasing the amount of bolts is therefore expected to lower the latency as the increased processing capabilities reduce the chance of queue buildup within the system. In other words, tuples have a higher chance of being processed without delay by the available bolts.

  In order to study the impact of the number of bolts on the overall system performance, we fix all other parameters at 1 while varying the number of bolts at {1, 2, 4, 8, 16, 24, 32}. Again, Figure 1a illustrates the effect upon throughput (in terms of tuples per second) and Figure 1b the effect upon the latency (in terms of milliseconds to process a tuple) as the number of bolts is increased. An increase in bolt parallelization reduces throughput due to the extra CPU load associated with scheduling. However, a reduction in latency towards a lower limit is also observed when the number of bolts is within the {8, 24} range. This limit represents the fastest possible bolt execution time, which is the cost of the system from the emission of the tuple by the spout up to the completion of the algorithm implemented by bolt B.

Power consumption. This section quantifies how system configuration impacts CPU usage (and hence power consumption). With this experiment we demonstrate that CPU usage is not linearly dependent upon performance. For example, in **Figure 2,** we observe that the throughput for the configuration $< 1, 16, 1, 1 >$ is equivalent to the throughput for configuration $< 1, 8, 8, 16 >$, yet the CPU usage is almost doubled. This observation demonstrates the complexity and subtleties involved in efficient system configuration.

Horizontal scalability configuration. One of the benefits of a distributed system is its capacity to increase the number of parallel threads of execution and reliably distribute them over the cluster nodes so as to improve processing efficiency in time and capacity. However as shown above, tuning a configuration is a complex manual task that can involve several tests before the optimal configuration can be identified. While we have shown that it is rather complex to efficiently configure a cluster with a single node, setting up a (possibly heterogeneous) dynamic cluster with a large number of nodes can become an even more difficult operation. With Storm, the number of parallel threads of execution can be adjusted by tuning the number of spouts and bolts. As more nodes are added, the number of spouts can be safely

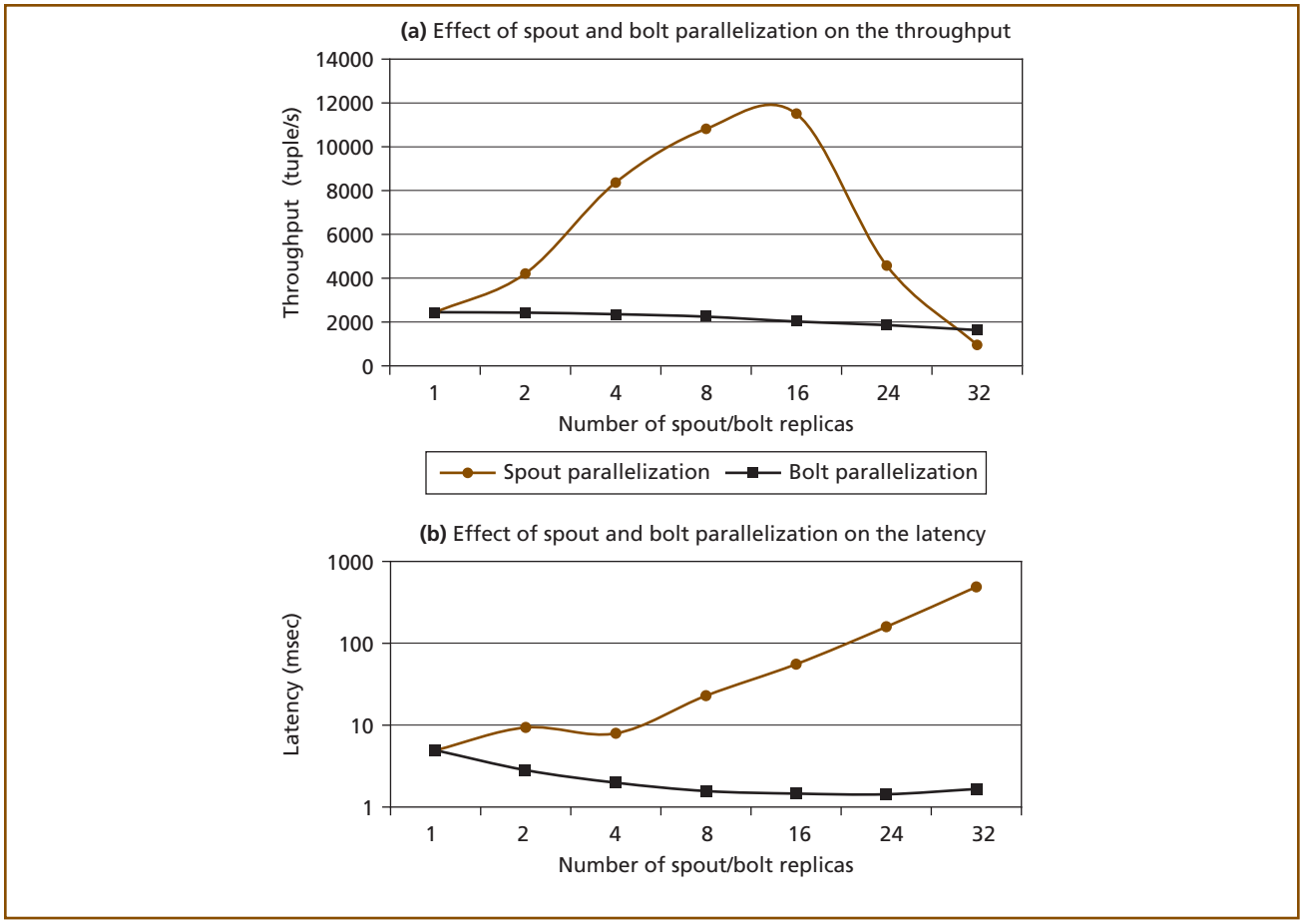

Figure 1.
Measuring latency and throughput on the different system configurations.

increased to some limit (i.e., before latency becomes a limiting factor as demonstrated previously). The number of bolts can be selected as a ratio of the number of spouts, for example with 100 spouts and a ratio of 0.2, we create 20 bolts. **Figure 3** shows the result of a series of experiments where we created topologies on clusters of multiple sizes with an increasing number of spouts. We tested the throughput on these topologies where the ratio of the number of bolts to number of spouts is adjusted. Figure 3 demonstrates that the optimum ratio is independent of the cluster size and number of spouts, however we have not tested whether it is independent of the specific topology. Nevertheless, this represents an important step in determining a method for optimally configuring large Storm clusters based on experiments using a smaller test cluster.

Parameter Configuration Engine: First Steps

In the previous section we demonstrated how the choice of configuration parameters profoundly impacts the performance of Storm. However, it is almost impossible to determine a priori which configuration parameters are best. Our experiments have demonstrated that the configuration of different parameters impacts throughput and latency in different ways. To better understand this, we create insightful cost models [8] which better describe the interplay between these factors and how they impact data flow, data processing, and system management.

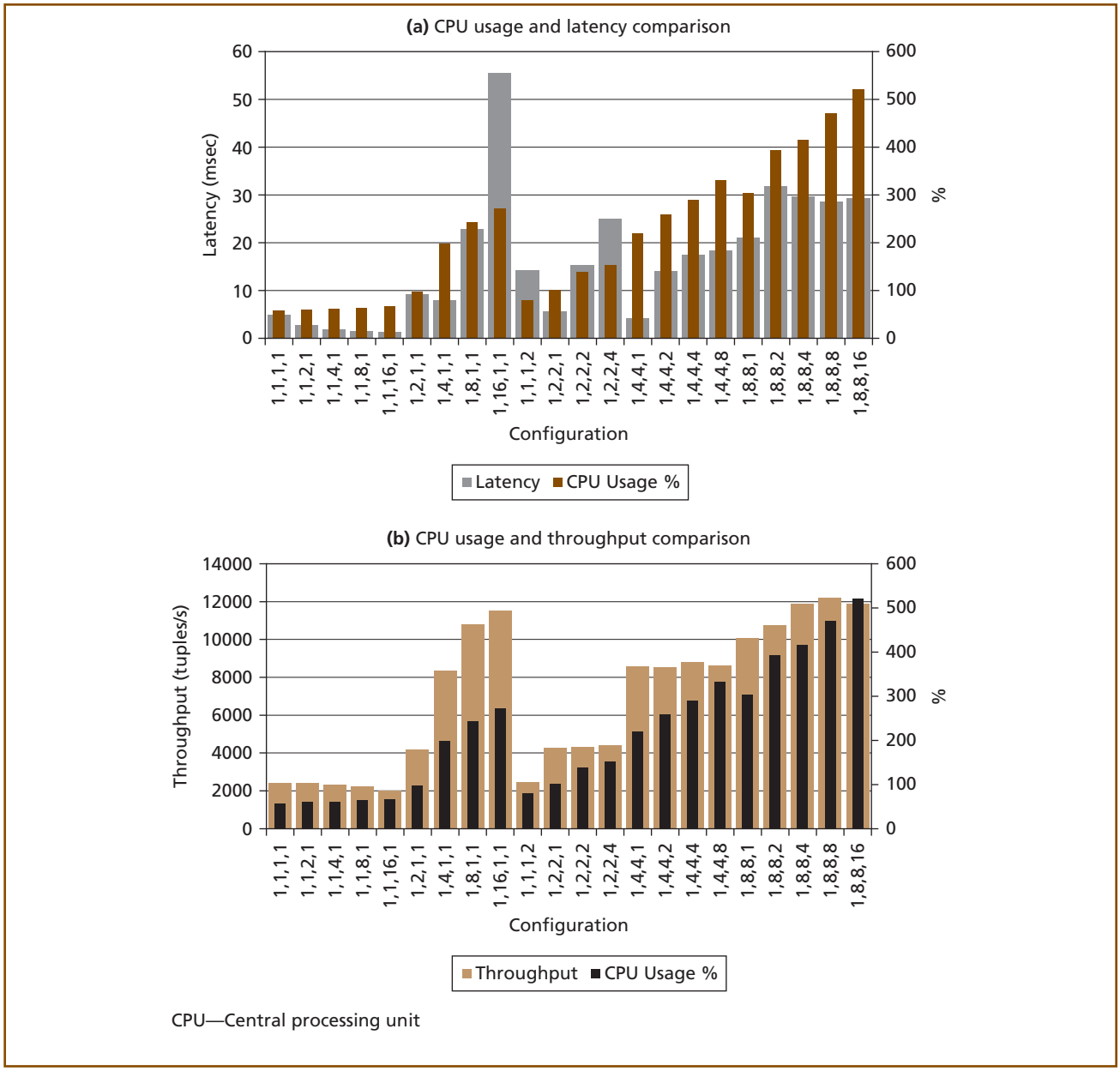

Figure 2.
Comparing throughput and latency with CPU usage percentage.

Our long term goal is to create a configuration engine based on these models which will enable optimal running of Storm. This will be achieved based on a search of the parameter space, and the insights provided by the cost models. In this paper, we present the first steps to creating this engine by providing a deep description of Storm's inner workings via these cost models.

Data Flow Cost Model

This section presents the data flow cost model used to characterize the size (in bytes) of data flowing through an active Storm topology and the cost (in seconds) of transferring data between the processing components (i.e., spouts and bolts). In the Storm framework, data are assumed to be made available by an external source (e.g., message queue or file

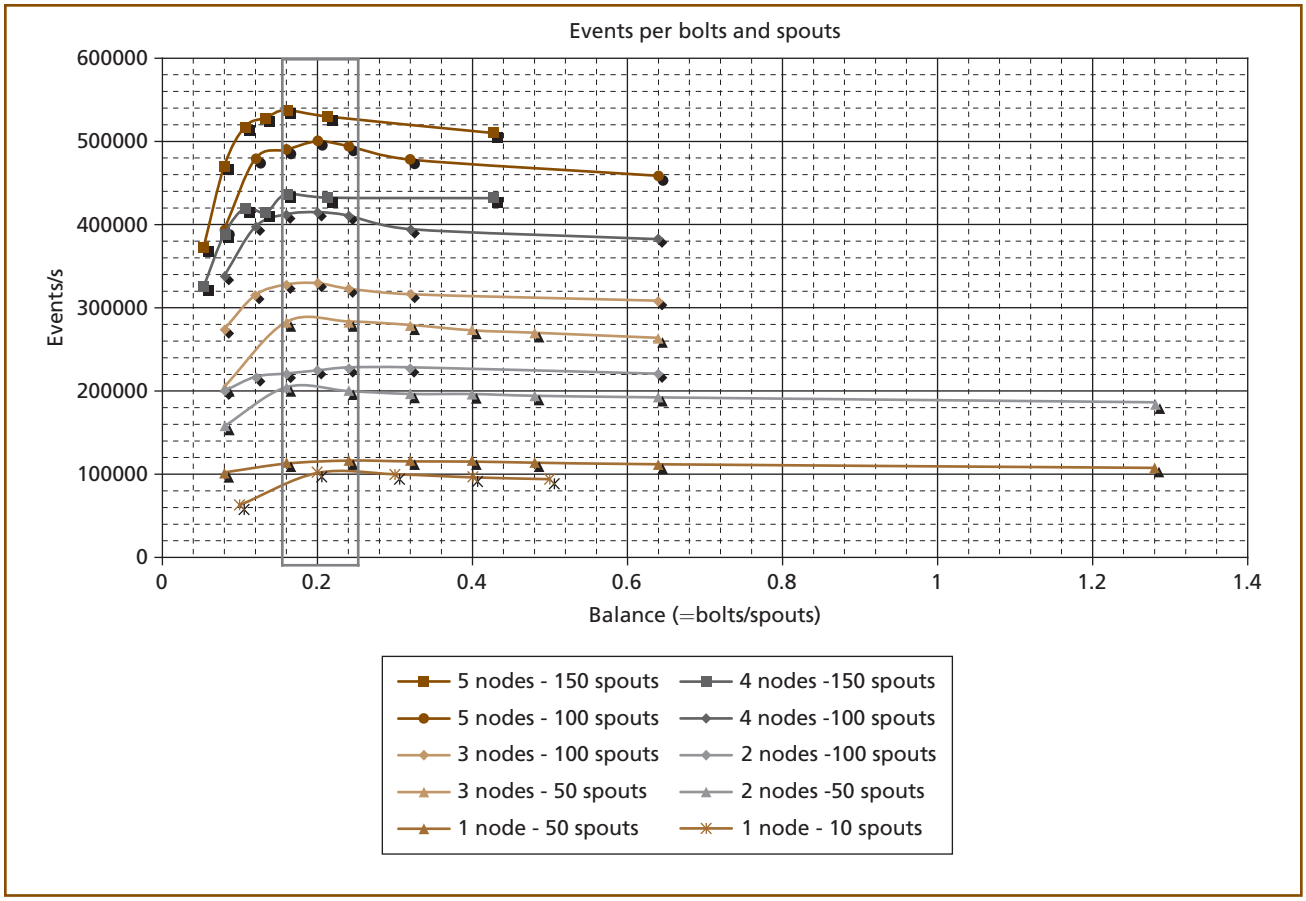

Figure 3.
Horizontal scalability optimal configuration range.

system) and read by a spout. The spout then transforms the data from its raw format into tuples, i.e., the internal scheme. These tuples are then emitted into the Storm cluster according to a chosen partitioning algorithm (i.e., stream grouping). This way, tuples are sent through the computational components, i.e., the bolts. While tuples flow through the Storm topology, they may get merged, split, and transformed several times.

Data size. We compute the data size that flows in a Storm topology as the amount of data that enters the system in the form of tuples which are injected into the Storm cluster by the various spouts plus the tuples which are generated and emitted by the various processing bolts and are consumed by other processing bolts.

- *Input-output data size.* Considering that a spout can generate input tuples of different types, the data size of an input spout per time unit can be computed as the sum of all tuples arriving per time unit, multiplied by their respective byte size (according to the specific tuple type). Similar to the input data size for the spout, we can compute the corresponding output data size of the bolts. Each processing bolt in a topology consumes at least one stream of tuples and possibly generates and emits one or more new streams of tuples. Therefore, the number of output tuples in a processing bolt is a function of the number of received tuples and their types per time unit.
- *Topology data flow size.* Finally, the total data flow size for a generic Storm topology can then be

expressed as the total data input size injected by all spouts into the topology added to the total data output size emitted by all the processing bolts in the topology. Therefore, we can formalize the total data flow as follows:

$$Flow.Size[bytes] = \sum_{i=1}^{|S|} Injected.Size_i + \sum_{j=1}^{|B|} Emitted.Size_j$$

where S is the set of spouts and B refers to the set of bolts which are instantiated in the executed topology.

Data transfer cost. The data transfer cost is the cost (in seconds) of actually delivering the tuple to a destination task (i.e., bolt instance). A distinction must be made between various allocations of communicating spouts and bolts across the cluster because running tasks remotely versus locally produces substantially different communication costs in terms of bandwidth consumption and communication time, i.e., latency. Therefore, we distinguish between three categories of data transfer cost: within the same Java virtual machine (JVM), within the same node but in different JVMs, and across different cluster nodes. In the case where the tuples are passed between spout-bolt or bolt-bolt in the same JVM, the tuples are immediately placed into the consuming bolt's receive queue without any manipulation of their data format. In contrast, when tuples are exchanged between different JVMs in the same cluster node, a serialization/deserialization cost is added to send the tuple from one JVM to another. Finally, when communicating spout-bolts or bolt-bolts are allocated on different JVMs hosted on different cluster nodes, the added costs are generated both from the serialization/deserialization step and from network transfer cost.

Data Processing Cost Model

The data processing cost model highlights the execution time computed as the composition of the spout and the bolt processing costs (both in units of seconds). Specifically, the spout processing cost represents the cost of reading a raw event from an unspecified source and injecting a Storm tuple into the topology. The bolt processing cost, on the other hand, represents the cost of processing a tuple in a bolt and possibly emitting new tuples into the topology for further processing. Note that the bolt processing cost is fundamentally affected by the computational complexity of the algorithmic intelligence implemented in the bolt logic.

- *Spout CPU processing cost.* The CPU processing cost of a spout (S) is a function of the tuple emit rate, where consideration must be made for possibly having multiple concurrent tuple types (t), each accounting for slightly different processing costs. Specifically, there is a CPU cost to read a raw input tuple of type t and a CPU cost of transforming a raw input tuple of type t into a Storm tuple of type t. Finally, there is the cost of partitioning the input tuples of type t which varies according to the stream grouping algorithm (*shuffle, fields, all, global*). There is also a serialization cost if the bolt is hosted on a different JVM.
- *Bolt CPU processing cost.* There are two sequential phases tuples go through while being processed by a bolt: a *transform* phase that accounts for tuple transformation steps, like serialization/deserialization, and a more general execute phase in which the actual bolt's processing logic is performed. Therefore the processing cost of a bolt is a function of the number of tuples (of type t) received by the bolt per time unit, plus the cost to generate (if any) new tuples (of type t') in response to receiving input tuples (of type t). It is clear that the major cost for processing a bolt will be determined by the algorithmic intelligence implemented in the bolt.
- *Topology CPU costs.* The total processing cost on the entire Storm topology is represented as the sum of the total CPU processing cost for all of its spouts and bolts. Therefore, the topology cost, i.e., *T:CPUCost*, for a Storm topology is defined as:

$$T.CPUCost[\text{seconds}] = \sum_{i=1}^{|S|} Spout.Cost_i + \sum_{j=1}^{|B|} Bolt.Cost_j$$

where S and B represent the sets of spouts and bolts in the topology.

System Management Cost Model

The system management cost models the impact of the set of tasks, abstracted away from the user, which are required to run a Storm cluster. These

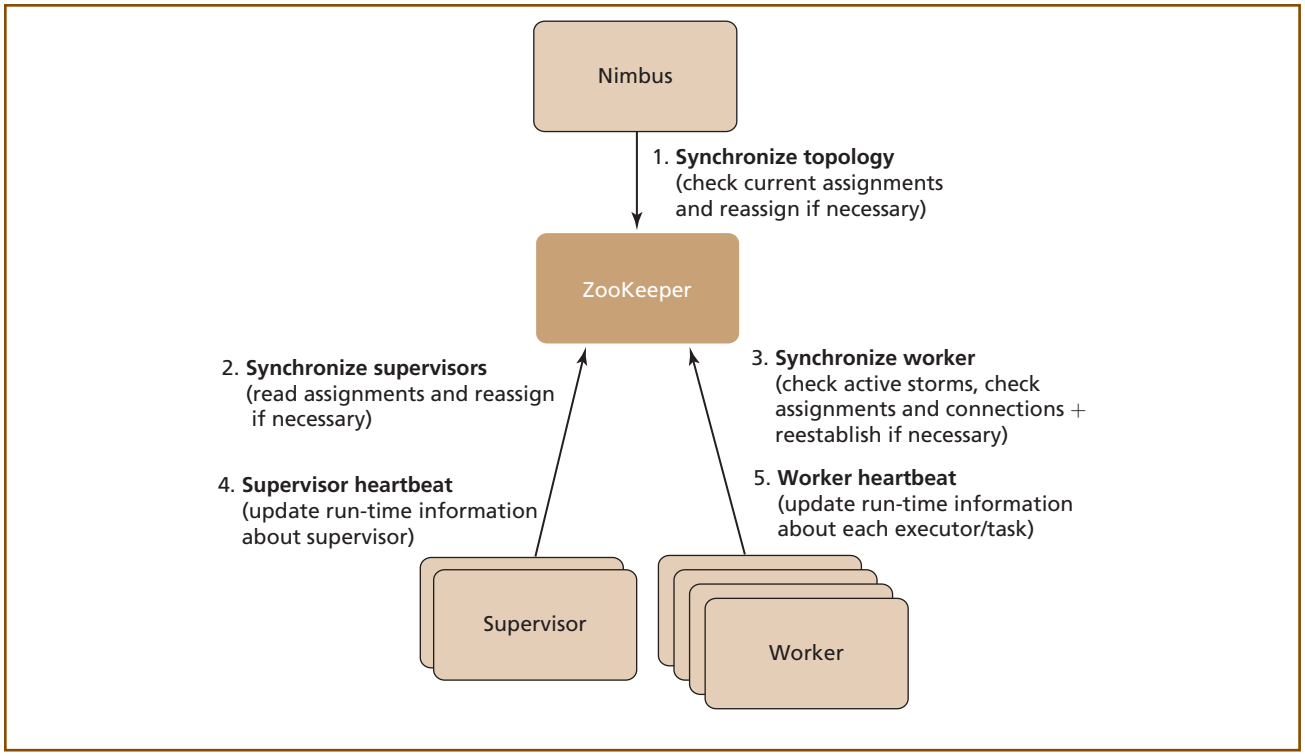

Figure 4.
ZooKeeper interactions.

tasks include provision of support for node failure/addition/removal, JVM failures, and network issues. Storm's system management tasks are coordinated through ZooKeeper and are mainly related to the interaction between ZooKeeper and Nimbus, supervisor and worker, as shown in **Figure 4**.

There are five recurring system management tasks that interact with ZooKeeper:

1. *Synchronize topology.* Nimbus checks the active assignments and compares them to the required assignments according to the topology specification. If a difference is detected, e.g., because of node failure, Nimbus will reassign the unassigned tasks over the available worker processes in the cluster.
2. *Synchronize supervisors.* Each supervisor reads its assignments from ZooKeeper and reassigns them if it detects a difference between what it has currently assigned across its workers. Reassignment takes the form of updates to ZooKeeper's assignments for the workers to query during their next poll cycle.
3. *Synchronize workers.* Each worker reads its assignments from ZooKeeper. If there is a mismatch, the missing connections are established. In addition to this, each worker also checks the active Storm topologies. If the Storm topology for which it is running tasks is no longer active (because it was explicitly killed), the worker would need to stop processing.
4. *Supervisor heartbeat.* Each supervisor will send a heartbeat to ZooKeeper. A heartbeat takes the form of storing some run-time information about the supervisor in ZooKeeper.
5. *Worker heartbeats.* Similarly, each worker sends a heartbeat to ZooKeeper. A worker heartbeat includes statistical information describing each task running in that worker.

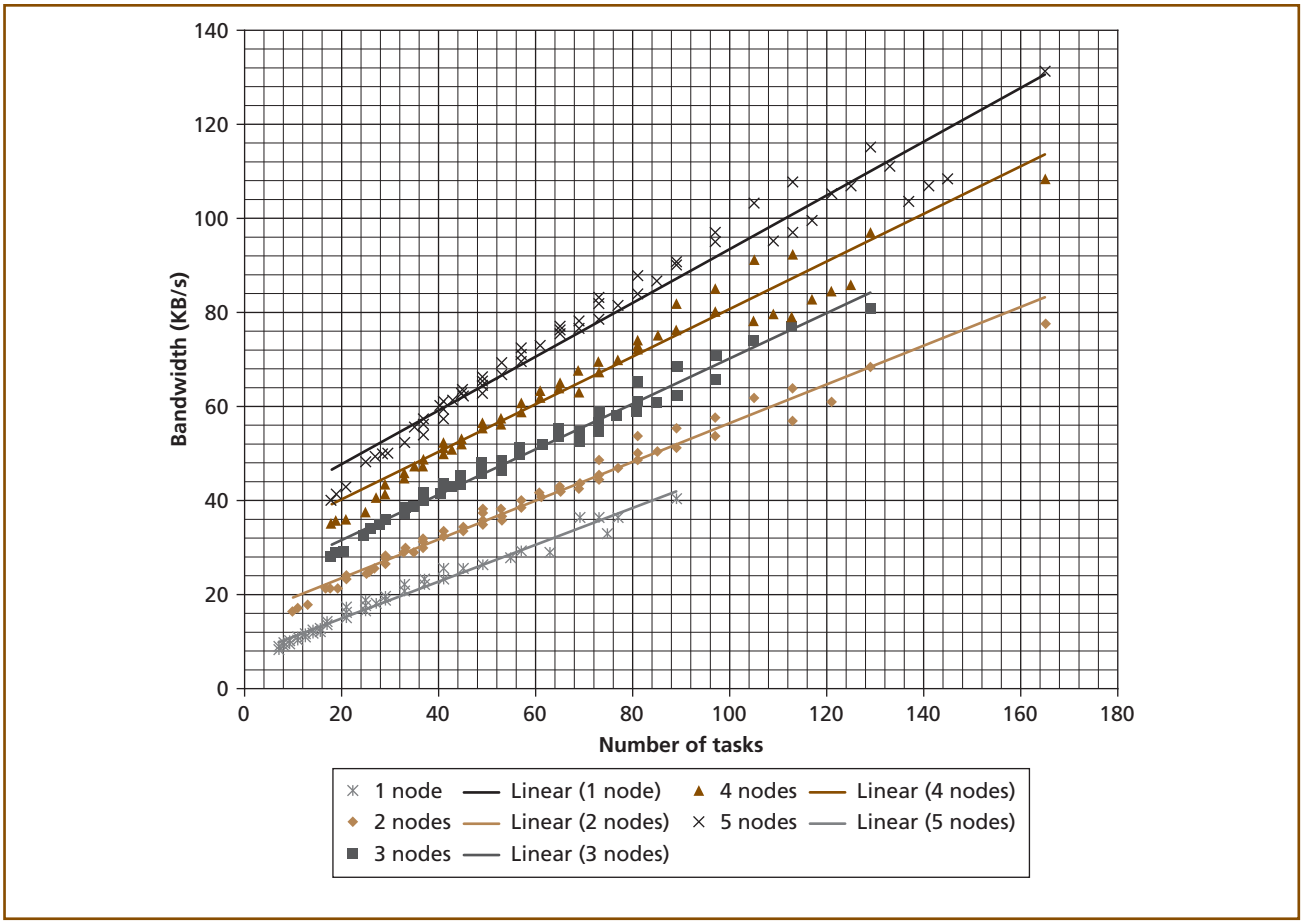

Figure 5.
System management traffic as a function of the number of components.

In principle, the system management cost has two components: a component that reflects the load on ZooKeeper, expressed in the number of requests per time unit and a network load component which represents the network bandwidth consumption.

From the experimental evaluation reported in **Figure 5**, we can summarize that ZooKeeper traffic bandwidth increases proportionally with the number of tasks and nodes running in the cluster. In the figure we plot the measured bandwidth (in KB/s) attributed to ZooKeeper communication as a function of the number of tasks running in the cluster, for various cluster sizes. Note also that ZooKeeper communication is independent of the specific algorithms implemented in spouts or bolts and independent of the event input rate. Overall, the system management cost (in seconds) represented less than 10 percent of the cost of running our experimental Storm topology.

Building the Parameter Configuration Engine

In the first half of this paper we demonstrated that optimal configuration of Storm is non-trivial. Increasing parallelization profoundly impacts throughput and latency in a manner which is difficult to determine a priori. As a first step to building a configuration engine for Storm, we provided insightful models into the fundamental data flow, data processing, and system managements costs which impact throughput and latency. Our plan is to develop an engine to

leverage these models which, along with real time input from a running Storm system, will select a set of configuration parameters to deliver optimal performance.

Related Work

The growing demand for large-scale data processing and data analysis applications has spurred the development of novel solutions from both industry (e.g., web data analysis, click-stream analysis, network-monitoring log analysis) and the sciences (e.g., analysis of data produced by massive scale simulations, sensor deployments, and high-throughput lab equipment). MapReduce [12] is a framework which was introduced by Google for programming commodity computer clusters to perform large-scale data processing. The framework is designed such that a MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner. However, the basic architecture of the MapReduce framework requires that the entire output of each respective map and reduce task be materialized into a local file and Hadoop Distributed File System (HDFS), before it can be consumed by the next stage. Therefore, it is not adequate for supporting real time processing of streaming data.

Several approaches have been proposed to tackle this challenge. For example, the MapReduce online approach [11] has been proposed as a modified architecture of the MapReduce framework in which intermediate data is pipelined between operators while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. The Incoop system [9] has been introduced as a MapReduce implementation that has been adapted for incremental computations which detect changes to input datasets and enable the automatic update of the outputs of the MapReduce jobs by employing a fine-grained result reuse mechanism. In particular, this allows MapReduce programs which have not been designed for incremental processing to be executed transparently in an incremental manner. The Main-Memory MapReduce (M^3) system [6] has been proposed to support the answer of continuous queries over streams of data bypassing the HDFS so data is processed only through a main-memory-only data path and totally avoids disk access. In this approach, mappers and reducers never terminate where there is only one MapReduce job per query operator that is continuously executing.

While the above approaches are bringing MapReduce closer to a stream processing system and have the added benefit of requiring only a single programming model to support both batch and streaming use cases, event stream processing systems have been designed from the ground up to support streaming use cases; they are not hampered by limitations or design decisions that were made in light of a totally different context. Moreover, the MapReduce paradigm seems to be counterintuitive to implementing sizeable topologies of processing nodes to handle complex analytical algorithms on streaming data. While MapReduce solves complex calculations through pipelining multiple map and reduce stages in a flow, an event stream processing system starts from a single input operator (which could be compared to a map stage), followed by a sequence of processing steps. At each step, a range of partitioning algorithms can be chosen to distribute output events over the available instances of the next processing step in the flow, allowing for a more natural design.

Several distributed stream processing systems have been presented in the literature. They include Aurora [2], Borealis [1], the Stream Processing Application Declarative Engine (SPADE) [13], Stormy [19] and Apache S4 [7]. For example, in Borealis, the collection of continuously running queries is treated as one giant network of operators. The processing of these operators is distributed to multiple sites where each site runs an instance of the Borealis server. The query processing is controlled by an admin component which takes care of moving query diagram fragments to and from remote Borealis nodes when instructed to do so by other components. SPADE is a declarative stream processing engine which supports a set of basic stream-relational operators with powerful windowing and punctuation semantics. The system is designed to execute a large number of long-running jobs that take the form of data flow graphs where each graph consists of a set

of processing elements connected by streams and each stream carries a series of stream data objects.

The processing elements can communicate with each other via their input and output ports. The concepts and ideas of our proposed models can be easily adopted to be used within the context of these systems.

Giurgiu [14] has presented an approach for estimating the performance of mobile-cloud applications. The approach tried to identify the factors that impact interaction response times, such as the application distribution schemes, workload sizes and intensities, or the resource variations of the mobile-cloud application setup. It also attempted to find correlations between these factors in order to better understand how to build a unified and generic performance estimation model. The Starfish system [15, 16] is the most relevant system for our work. It represents a cost-based optimizer for MapReduce programs which focuses on the optimization of configuration parameters for executing these programs on the Hadoop platform. It relies on a profiler component that collects detailed statistical information from executing the programs and a what-if engine for fine-grained cost estimation processing. For a given MapReduce program, the role of the cost-based optimizer component is to enumerate and search efficiently through the high dimensional space of configuration parameter settings, making appropriate calls to the what-if engine, in order to find the optimal configuration setting. It clusters parameters into lower-dimensional subspaces such that the globally-optimal parameter setting in the high-dimensional space can be generated by composing the optimal settings found for the subspaces. Our current study represents the first step in the implementation of a similar what-if analyzer component in the more complex environment of real time distributed stream-processing engines.

Conclusion

Batch computation systems such as Hadoop have greatly aided the analysis of large datasets in the telecom ecosystem. The scalable, fault-tolerant nature of this framework has made it an important tool in distributed computation. However, adapting in real time to the rapidly changing conditions in a telco network requires real time streaming analysis that can be run without a complete dataset. Streaming analytics can prove beneficial for the rapid response to performance data, fault data, and customer experience data which can indicate network issues and customer satisfaction issues. In this paper we have provided a set of use cases for real time streaming analytics in the telecom ecosystem. We have demonstrated the inherent difficulties in efficiently tuning configuration parameters in Storm and we used these difficulties as a motivator to create a parameter configuration engine. As a set of first steps towards the creation of such an engine, we performed a deep analysis of the inner workings of Storm and created models for data flow, CPU cost, and system management cost.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, 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," Proc. 2nd Biennial Conf. on Innovative Data Syst. Res. (CIDR '05) (Asilomar, CA, 2005), pp. 277–289.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik, "Aurora: A Data Stream Management System," Proc. ACM SIGMOD Internat. Conf. on Management of Data (SIGMOD '03) (San Diego, CA, 2003), Demonstration, p. 666.

[3] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "StreamKM++: A Clustering Algorithm for

Data Streams," ACM J. Exp. Algorithmics, 17:2 (2012), article 2.4.

[4] R. Ahas, M. Tiru, and A. Kuusik, "Measuring Repeated Visitations with Mobile Positioning Data. Applications for Marketing," Proc. 2nd Conf. on the Analysis of Mobile Phone Datasets and Networks (NetMob '11) (Cambridge, MA, 2011).

[5] Alcatel-Lucent, "9360 Small Cells," <http://www.alcatel-lucent.com/products/9360-small-cell>.

[6] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, "M^3: Stream Processing on Main-Memory MapReduce," Proc. 28th IEEE Internat. Conf. on Data Eng. (ICDE '12) (Washington, DC, 2012), pp. 1253–1256.

[7] Apache Software Foundation, "S4 Distributed Stream Computing Platform," <http://incubator.apache.org/s4/>.

[8] I. Bedini, S. Sakr, B. Theeten, A. Sala, and P. Cogan, "Modeling Performance of a Parallel Streaming Engine: Bridging Theory and Costs," Proc. 4th ACM/SPEC Internat. Conf. on Perform. Eng. (ICPE '13) (Prague, Cze., 2013), pp. 173–184.

[9] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini, "Incoop: MapReduce for Incremental Computations," Proc. 2nd ACM Symp. on Cloud Comput. (SOCC '11) (Cascais, Prt., 2011), article no. 7.

[10] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure, "Experience Report: Processing 6 Billion CDRs/Day: From Research to Production," Proc. 6th ACM Internat. Conf. on Distrib. Event-Based Syst. (DEBS '12) (Berlin, Ger., 2012), pp. 264–267.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," Proc. 7th USENIX Symp. on Networked Syst. Design and Implementation (NSDI '10) (San Jose, CA, 2010), pp. 313–328.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proc. 6th Symp. on Operating Syst. Design and Implementation (OSDI '04) (San Francisco, CA, 2004), pp. 137–150.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," Proc. ACM SIGMOD Internat. Conf. on Management of Data (SIGMOD '08) (Vancouver, BC, Can., 2008), pp. 1123–1134.

[14] I. Giurgiu, "Understanding Performance Modeling for Modular Mobile-Cloud Applications," Proc. 3rd ACM/SPEC Internat. Conf. on Perform. Eng. (ICPE '12) (Boston, MA, 2012), pp. 259–262.

[15] H. Herodotou, F. Dong, and S. Babu, "MapReduce Programming and Cost-Based Optimization? Crossing This Chasm with Starfish," Proc. VLDB Endowment, 4:12 (2011), 1446–1449.

[16] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-Tuning System for Big Data Analytics," Proc. 5th Biennial Conf. on Innovative Data Syst. Res. (CIDR '11) (Asilomar, CA, 2011), pp. 261–272.

[17] C. Kaiser and A. Pozdnoukhov, "Modelling City Population Dynamics from Cell Phone Usage Data Streams," Proc. 2nd Conf. on the Analysis of Mobile Phone Datasets and Networks (NetMob '11) (Cambridge, MA, 2011), pp. 69–71.

[18] P. Krivitsky, P. Ferreira, and R. Telang, "Network Neighbor Effects on Customer Churn in Cell Phone Networks," Proc. 2nd Conf. on the Analysis of Mobile Phone Datasets and Networks (NetMob '11) (Cambridge, MA, 2011), pp. 112–116.

[19] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An Elastic and Highly Available Streaming Service in the Cloud," Proc. EDBT/ICDT Joint Conf.: 15th EDBT Internat. Conf. on Extending Database Technol., and 15th ICDT Internat. Conf. on Database Theory (EDBT/ICDT '12) (Berlin, Ger., 2012), pp. 55–60.

[20] W. Pan, N. Aharony, and A. Pentland, "Composite Social Network for Predicting Mobile Apps Installation," Proc. 2nd Conf. on the Analysis of Mobile Phone Datasets and Networks (NetMob '11) (Cambridge, MA, 2011), pp. 100–102.

[21] Storm, <http://storm-project.net/documentation.html>.

[22] V. Yeshwanth and M. Saravanan, "Churn Analysis in Mobile Telecom Data Using Hybrid Paradigms," Proc. 2nd Conf. on the Analysis of Mobile Phone Datasets and Networks (NetMob '11) (Cambridge, MA, 2011), pp. 105–108.

(Manuscript approved August 2013)

BART THEETEN is a member of technical staff at Bell Labs in Antwerp, Belgium. He holds an M.Sc. in computer engineering from the University of Ghent. He began his career as a software engineer working on various network and element management solutions in various locations throughout the company, including Alcatel USA in Raleigh, North Carolina and Alcatel CIT in France, before joining Bell Labs in Antwerp, where he specializes in service-oriented architectures, service/application enablement technologies, and web services.

IVAN BEDINI was a member of technical staff at Bell Labs in Blanchardstown, Ireland while this research was conducted and the paper prepared for publication. He is currently team leader for the BigData project at Trento RISE, an EIT ICT Labs partner founded by the University of Trento and the Bruno Kessler Foundation. The aim of Trento RISE is to contribute to the creation of a center of excellence for research, innovation, and advanced training in the field of information and communications technology. Dr. Bedini is responsible for developing innovative solutions for the collection, integration, and analysis of big data, as well as the creation of a big data services platform. During his tenure at Bell Labs, he was an active contributor to the Semantic Data Access (SDA) and BigData analytics projects. Prior to joining Bell Labs, he spent 10 years as a researcher with Orange Labs France in the Enterprise Applications and the Trust & Secure Transactions research departments. He has also four years of contribution to standardization bodies as member of the UN/CEFACT Information Content Management Group and as a member of the OASIS ebXML Registry/Repository Committee. He received his Ph.D. from the University of Versailles, France. He is an expert in information extraction, data integration, semantic technologies, big data technologies, and distributed computing.

PETER COGAN was a member of technical staff at Bell Labs in Blanchardstown, Ireland while this research was conducted and the paper prepared for publication. He is currently a senior data scientist with Changing Worlds at Amdocs. Changing Worlds is a University College Dublin spinout specializing in data analytics and recommendation which was acquired by Amdocs in 2008. At Amdocs, Dr. Cogan is conducting research and development into machine learning and big data in the Telco space. Prior to his appointment at Amdocs, Dr. Cogan was a member of technical staff at Bell Labs in Dublin, Ireland. At Bell Labs Dr. Cogan conducted research into diverse big data projects related to workforce planning, social networks, CDR analytics, proactive care and real time machine learning. He completed both his primary degree in experimental and mathematical physics and his Ph.D. in high energy astrophysics at University College Dublin. During his Ph.D., he specialized in analysis techniques for nanosecond sampling of atmospheric Cerenkov radiation and applied these techniques to analysis of gamma-ray observations of blazars. He was appointed as a postdoctoral researcher at McGill University, Montreal, Canada where he worked for two years prior to joining Bell Labs.

ALESSANDRA SALA is the technical manager of the Data Analytics and Operations Research group at Bell Labs in Dublin, Ireland. In her prior appointment, she held a research associate position in the Department of Computer Science at the University of California Santa Barbara. During this appointment, she was a key contributor to several research proposals funded by the U.S. National Science Foundation and she received the Cisco Research Award in 2011. She focused her research on modeling massive graphs with an emphasis on privacy threats for online social network users. Before that, she worked for two years as post-doctoral fellow with the CurrentLab research group led by Prof. Ben Y. Zhao. She completed her Ph.D. in computer science at University of Salerno, Italy. Her research interests include distributed algorithms and complexity analysis with an emphasis on graph algorithms and privacy issues in large-scale networks. Currently, she is investigating massive call detail record (CDR) data and large-scale graphs, like online social networks, to extract and combine valuable information from heterogeneous data sources. Ultimately, her research aims to design private solutions to access and analyze users' sensitive information while preserving their privacy.

TOMMASO CUCINOTTA is a member of technical staff at Bell Labs in Dublin, Ireland. He has a computer engineering degree from the University of Pisa, as well as a Ph.D. in computer engineering from the Scuola Superiore Sant'Anna of Pisa, where he spent more than 10 years researching topics including computer security, smartcard-based authentication,

soft real time scheduling, embedded systems, resource management, general purpose operating systems, real time virtualization and cloud computing. He has published several scientific papers on the topics above and serves as reviewer for several international journals, conferences, and workshops in his areas of expertise. ◆