High Performance CDR Processing with MapReduce

Mulya Agung* & A. Imam Kistijantoro

School of Electrical Engineering and Informatics,
Institut Teknologi Bandung, Jalan Ganesha No. 10, Bandung 40132, Indonesia
*E-mail: agung@tritronik.com

Abstract. A call detail record (CDR) is a data record produced by telecommunication equipment consisting of call detail transaction logs. It contains valuable information for many purposes in several domains, such as billing, fraud detection and analytical purposes. However, in the real world these needs face a big data challenge. Billions of CDRs are generated every day and the processing systems are expected to deliver results in a timely manner. The capacity of our current production system is not enough to meet these needs. Therefore a better performing system based on MapReduce and running on Hadoop cluster was designed and implemented. This paper presents an analysis of the previous system and the design and implementation of the new system, called MS2. In this paper also empirical evidence is provided to demonstrate the efficiency and linearity of MS2. Tests have shown that MS2 reduces overhead by 44% and speeds up performance nearly twice compared to the previous system. From benchmarking with several related technologies in large-scale data processing, MS2 was also shown to perform better in the case of CDR batch processing. When it runs on a cluster consisting of eight CPU cores and two conventional disks, MS2 is able to process 67,000 CDRs/second.

Keywords: *call detail records; Hadoop; high performance; Java EE; MapReduce; telecommunication mediation.*

1 Introduction

CDRs generated by low-level telecommunication equipment must first be prepared before they can be processed by high-level applications. In telecommunications, the system that handles this preprocessing stage is called the mediation system. Due to the large size of the generated CDRs and the need for fast processing of the results for various purposes, the preprocessing stage in the mediation system is a big data challenge. As discussed in [1], to achieve acceptable performance, this kind of application needs other techniques than conventional computation.

For many years, a scalable mediation system called MS1 has been used and running in production systems to perform CDR processing for one of the biggest telecommunication providers in Indonesia. However, due to the fast growth of subscribers, using it at the current scale becomes increasingly

Received October 1st, 2015, 1st Revision November 2nd, 2015, 2nd Revision February 2nd, 2016, Accepted for publication March 14th, 2016.
 ISSN: 2337-5787, DOI: 10.5614/itbj.ict.res.appl.2016.10.2.1

challenging. The biggest challenge comes from efficiency, i.e. limited resources are demanded to process more data.

In this paper, we present an investigation and analysis of the current parallel system overhead due to I/O bottlenecks. Then we present the design and implementation of a new system, which addresses the issues. The new system is called MS2. It runs on a Hadoop cluster environment and uses MapReduce for parallel data processing. We also present the results from an empirical evaluation and COST analysis [2] to determine the system overhead compared to single-thread performance. In the next section, we begin by briefly describing MS1 and the motivation for developing MS2. Then, in the Section 3, we describe an alternative that we considered to address the problems described above. Section 4 presents the analysis, design and implementation of MS2. The results from the empirical evaluation comparing MS1 and MS2 are presented in Section 5. In Section 6 we discuss related works and benchmark MS2 with several related technologies in big data processing. Finally, Section 7 contains our conclusion and points to some directions for future work.

2 Motivation for MS2

MS1 has served the CDR processing needs from telecommunication billing systems for the past several years. Based on our operation and maintenance experience at the current scale, we wanted to improve the performance of the current production system. Before that, we also needed to identify and analyze the possible bottlenecks or inefficiencies of MS1, which are highlighted in the following sections. These inefficiencies motivated us to develop MS2.

2.1 CDR Processing in Mediation System & MS1 Architecture

Due to efficiency constraints, CDRs are generated by low-level telecommunication equipment in a compact binary format [3]. This binary format should be converted to a textual format so it can be processed by next steps or higher-level applications easier. Mediation is the first step in processing the CDRs, which involves capturing CDRs from upstream network systems and making them ready for processing by downstream applications (RA, BI, FM, warehousing). This complex task is composed of several steps, including functions for collection, validation, filtering, collating, correlation, aggregation, formatting, normalization and data transformation [4].

MS1 splits one-pipeline processing into three stages. The collection and decoding stage involves collecting CDR files from external input sources and decoding them to an intermediary format. The preprocessing stage evaluates rules against the records' contents. Mediation functions such as validation,

aggregation, filtering and normalization are specified in these rules. The formatting and distribution stage converts and distributes the processed data to a destination-friendly format.

MS1 is designed and implemented by adapting the Java Enterprise Integration Patterns architecture (EIP) [5] to integrate its components. It uses a message-passing and shared-memory model for data parallelism and achieves distributed processing by moving data to computation. Processing components run in cluster nodes and data are distributed to computation nodes as needed (Figure 1).

2.2 MS1 Performance Measurement and Analysis & Efficiency Issues with MS1 Processing

The performance of the existing system was measured and analyzed by conducting two test scenarios: the first test was performed in a multi-thread configuration, the second was performed in a single-thread configuration. Both scenarios were tested on clustered nodes consisting of two identical servers (i.e. Server1 and Server2). The hardware specifications for each server were 2.66 GHz Intel Quad Cores CPU, 8 GB of DDR2 RAM, SATA 2 HDD 7200 RPM and one gigabit Ethernet. The servers ran on Linux Operating System. The performance parameters used for analysis were processing time and resources utilization.

Parallel performance was assessed using a 1.6 GB data set consisting 100 of 16 MB batch CDRs files. The numbers of threads allocated for decoding, preprocessing and formatting were 1, 8, and 20 respectively. The CPU utilization graphs (Figure 2) show a pattern which gives a low CPU utilization on average. The I/O utilization graphs show a similar pattern in which disk and ethernet load have a high peak load and are idle several times. The investigation continued by increasing the threads of the preprocessing stages. Figure 3(a) shows the performance of MS1 with several thread number configurations. The graph shows that increasing the threads does not increase performance significantly. The result of thread analysis using JVisualVM shows that the threads spend most of their time waiting for others to finish. There is no indication of deadlock or blocked threads caused by thread synchronization.

Many parallel systems have a surprisingly large overhead [2]. Their overhead is evaluated by benchmarking them against the performance of a single-thread system. For the same reason, this scenario was used for MS1. The single-thread configuration was tested on two types of I/O access, i.e. local storage access and remote storage access. The analysis of both cases gives an understanding of how both I/O types add more overhead to overall performance. The test results

showed that remote I/O gives more overhead than local I/O access. This overhead reduces the overall performance of MS1 significantly in the single-thread configuration and when using remote access (Figure 3(b)). The remote I/O overhead decreased the single-thread performance from 7658 record/s to 4427 record/s.

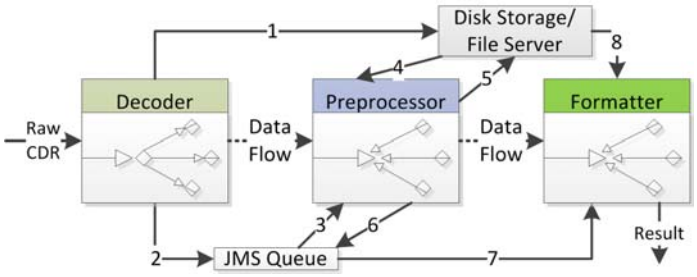

Figure 1 MS1 components and data flow.

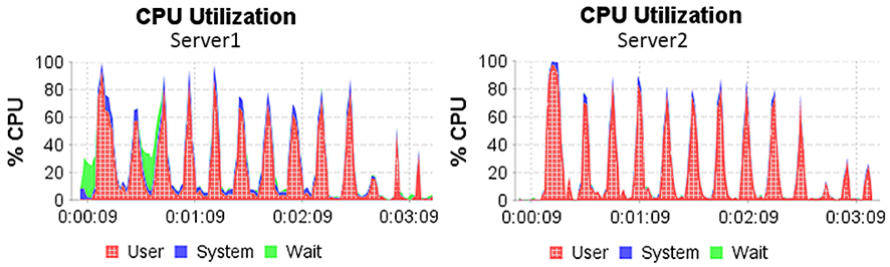

Figure 2 CPU utilization in MS1.

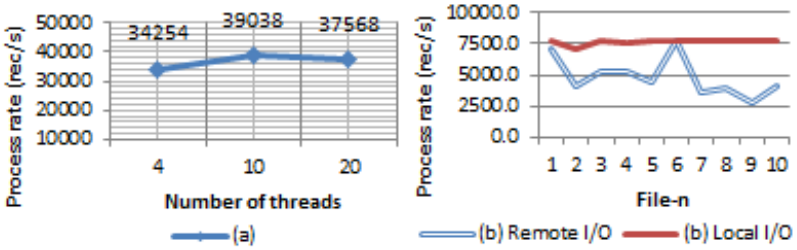

Figure 3 MS1 performance by (a) number of threads and (b) single-thread.

Based on the performance assessment of the parallel configuration and the single-thread configuration we concluded that the overall performance of MS1

is not optimum due to inefficient resources utilization. The inefficiency comes from I/O overhead caused by local disk access and network access. From the single-thread configuration test we found that the most significant reduction could come from network I/O. So by reducing these overheads, the overall performance of CDR processing should be increased significantly.

3 Design Alternative

As discussed in the previous section, the idea of increasing current performance is by reducing the I/O overhead. One principle that can be used to do this is locality [1,6]. This principle can minimize network I/O overhead by ensuring that computation is always performed against data stored in local nodes. The technique known to use this principle is moving computation to data. This technique is used in the MapReduce processing model to increase its batch processing performance [7]. Therefore, the analysis of MS1 brought us to the current hypothesis that adopting the MapReduce model to the existing CDR processing system will reduce the overhead of the distributed system and increase overall performance.

4 MS2

4.1 System Analysis

In this section, the systems analysis of MS2 is broken down based on identified problem topics. Data ingress is done by a collection and decoding process, while data egress is done by a formatting and distribution process. Several optimizations are done in this step, as explained in [8]. Aggregation of many and small data elements to one single big file can optimize data migration to HDFS. Data format transformation is needed because, in our case, CDR is not ideal to be directly processed by MapReduce. Based on the works [1,9], in fixed disk mode, sequential access has much better performance than random access. So in our case of using conventional disks, sequential pattern implementation will improve ingress performance.

Mapping existing components to the MapReduce model requires the following considerations. The preprocessing stage is more optimal when executed in a mapper process because of the large input data. Mapper will process the data using the locality principle [10,11] thus it will decrease network I/O access in large input CDR processing. The formatting stage is more optimal when it is executed in a reducer process because the formatter component gets intermediary data from the preprocessing result, which is much smaller than the ingress result. The reducer process uses intensive disk and network I/O, so to

decrease the overhead, the data and task amounts have to be as small as possible [11,12].

Transition from the existing architecture has its own challenges with regards to the MapReduce single fixed dataflow characteristic [13]. In the MapReduce environment, a Java Bean container that is initiated in one JVM cannot be accessed directly by Mapper and Reducer objects in another JVM. However, patterns such as Data Access Object (DAO), Singleton and Dependency Injection (DI) depend on this container. The solution for this problem is reinitiating the bean container when a Mapper and Reducer object is executed. Consequently, time and resources needed in the initiation process can impact the overall performance. An alternative and better solution is running the Java Bean Container outside of MapReduce and using a stateless remote procedure call to access the container [14].

The deployment strategy is important for the overall performance because it determines the total resources that will be utilized. To maximize resource utilization in existing nodes, HDFS and YARN should be deployed and running on all nodes. NameNode, DataNode, and NodeManager should run on all nodes with the replication level set to the number of nodes. Since there is only one global ResourceManager, its deployment can be chosen in one of the servers [10,15].

4.2 Design

Ingestor moves the CDR file to HDFS. It aggregates the data by multiplexing. It is also designed to run sequentially in order to improve disk access performance. To store the result of this aggregation process in a compressed big file, a new data structure was designed using the Avro serialization framework [8,10,11,16].

To maximize parallelism, the mapper component bundles partial decoding and preprocessing into one process. Mapper will take the ingested file based on batch key and decode binary contents to intermediary data. Then it will call RecordProcessor to evaluate the processing rules and generate an array of CdrData as its result. Mapper will access RefData service for every database lookup. There will be additional I/O from HTTP and database access, but this can be reduced by placing the database on the same host as the service. With this locality, mapper only needs network I/O to access the service. Database access performance is also improved with data caching [17]. Reducer executes formatting and serialization by generating records in a destination-friendly format. Serialization is needed to change the intermediary format to a text file

format that conforms to the upstream node. The reducer component is designed similar to the external output pattern, as explained in [18].

Before processing and formatting, CDRs generated by sources are loaded to MS2 by the ingestion process and stored in key-value based data (Figure 4). Under the MapReduce model, processing and formatting are expressed in terms of two processing stages (PS) – PS_{map} and PS_{reduce}. The same PS_{map} is running at all participating mappers, and the same PS_{reduce} is running at all participating reducers in parallel, such that each PS_{map} is applied to a set of key-value tuples (k,v) and transforms it into a set of tuples of a different type (k',v'); then all the values v' are re-partitioned by k' and each PS_{reduce} aggregates the set of values v' with the same k', as expressed in Eqs. (1) and (2).

$$PS_{map}: (k, v) => (k', v')* \quad (1)$$

$$PS_{reduce}: (k', v')* => (k', v'*) \quad (2)$$

Hadoop components are deployed to existing servers with the following structure. Node 1 and Node 2 are deployed in Server1 and Server2 respectively. Both HDFS and YARN components are deployed in both nodes. Primary NameNode and DataNode 1 of HDFS are deployed in Node 1. ResourcesManager and NodeManager 1 of YARN are also deployed in Node 1. Secondary NameNode, DataNode 2 and NodeManager 2 are deployed in Node 2. Figure 5 shows MS2 deployment in a Hadoop cluster and the execution of ingestion and the MapReduce process.

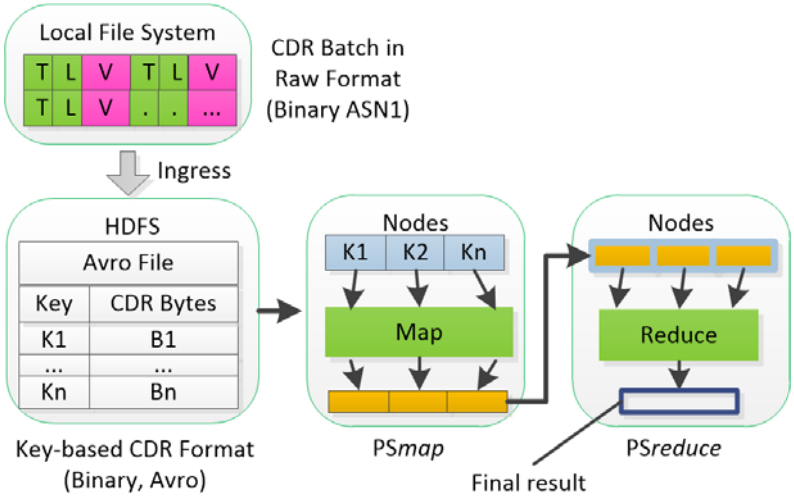

Figure 4 Data parallelism in MS2.

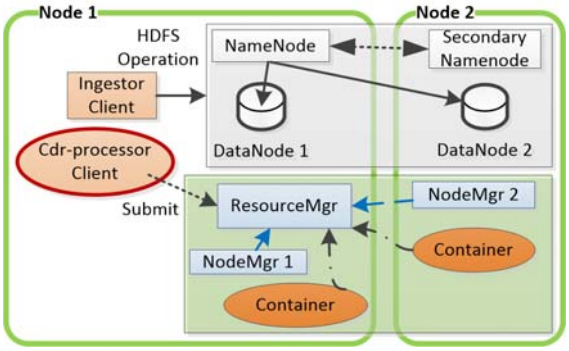

Figure 5 MS2 deployment in a Hadoop cluster.

4.3 Implementation

The new components implemented are Ingestor, Cdr-processor and intermediary data structure. Cdr-processor is a MapReduce driver consisting of Mapper and Reducer classes. The MapReduce platform used in this implementation is Hadoop 2.6.0 and is deployed and running with the same operating system as used in MS1. Besides the components written in Java, Hadoop provides components written in native C. These native components are needed to improve Hadoop performance by accessing resources directly [10]. The additional REST service component is implemented for reference database lookup. The initiation of a shared object which doesn't need I/O is implemented with the setup() method derived from Hadoop Mapper and the Reducer interface. By overriding this method, the initiation will be ensured to run once in the object's lifetime [10].

Compression is used to improve ingestion performance. Based on results from [12,19,20], snappy compression was chosen to give better performance than the built-in method. Some useful OS tunings recommended in [12,19] are applied in the deployed system. These include using the EXT4 file system and the noatime option set in mount parameters for mount point used by HDFS. The noatime option is useful to reduce disk-read overhead by disabling access log updates. The number of file descriptors allowed for the Hadoop process is also increased. This tuning is useful to prevent possible exceptions triggered at high loads, which may cause MapReduce jobs to fail.

Ingestion is performed to process the same data set used in MS1 performance assessment. To give more accurate results, this process is repeated for several

iterations. This stage runs in Node 1 and the duration was measured after each iteration was finished. Ingestion needs 36.8 seconds on average to collect a 1.6 GB data set and results in a 1.43 GB compressed file stored in HDFS.

The configurations tested to evaluate the processing performance were the Hadoop default configuration, tuning for 128 MB block size and tuning for 256 MB block size. From the test results, the configuration for 128 MB block size (Table 1) performed best with the highest CPU utilization (Figure 6). This configuration was set based on the recommended calculation in [12,19]. The input.fileinputformat.split.minsize property sets the split size allocated to the mapper process. It is set to 224MB so that if the input file resulting from ingestion has a size of 1.43 GB, the minimum number of splitted tasks will be 1.43 GB/224 MB = 6.38 ≈ 7 tasks. This number of tasks is enough for the total of 8 core CPU resources that the cluster has. If each core executes one mapper task, then overall the system still has one spare core for managing resources. Properties such as task.io.sort.mb, map.java.opts and map.memory.mb are set according to the number of split tasks and the total RAM available in the cluster. If one mapper task is allocated with 1280 MB heap, then one node will allocate (1280x4) MB = 5120 MB of memory. This number is still below the total RAM capacity of each node. This configuration ensures that mapper will not use swap memory to perform its tasks.

Table 1 Tuning configuration of 128 MB block size.

Configuration	Value
mapreduce.input.fileinputformat.split.minsize	224000000
mapreduce.task.io.sort.mb	500
mapreduce.map.java.opts	-Xmx1024m
mapreduce.map.memory.mb	1280

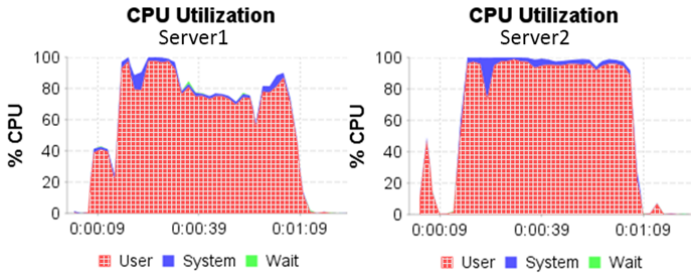

Figure 6 CPU utilization of 128 MB block size tuning.

Total performance is calculated from ingestion and Cdr-processor throughput. The performance of tested configurations is compared with MS1 and single-

thread performance (Figure 7(a)). MS2 with the 128-block size tuning configuration (MS2-128) had the best performance. The performance of the evaluated systems and CPU allocated to each configuration are shown in Table 2.

Compared to single-threaded MS1 (MS1-single-t), MS2 with 8 allocated cores resulted in 8.7x performance speed-up. The overheads of the compared systems were also measured. Figure 7(b) shows that the MS1 system had the highest overhead of the evaluated systems. To achieve 7658 rec/s performance, parallel MS1 needs 1.6 cores, whereas MS2-128 needs only 0.9 cores. This CPU cost reduction proves MS2-128 successfully reduced 44% of the overhead introduced by MS1. This result also shows that with the correct tuning, MS2 is able to achieve a performance that is more efficient than MS1, even compared with single-thread performance.

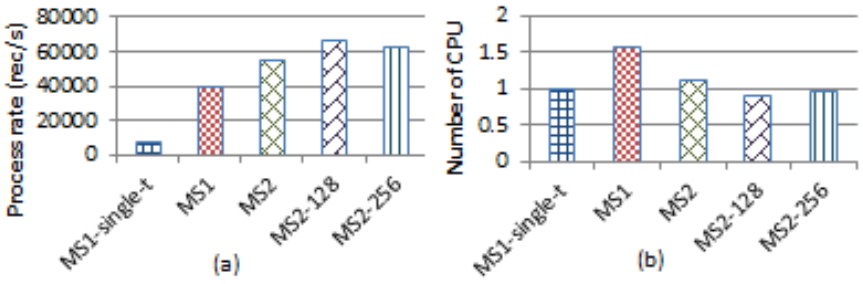

Figure 7 Comparison of (a) throughput (b) CPU to achieve single-t throughput.

Table 2 Performance of MS1 and MS2 by CPU.

System and configuration	Performance (rec/s)	CPU core
MS1 (10 preprocessing thread)	39037.57	8
MS2 (default Hadoop configuration)	54719.85	8
MS2-128 (block size=128, split task=7)	67060.70	8
MS2-256 (block size=256, split task=6)	62839.25	8
MS1 Single-thread	7658.73	1

Experiments were carried out to see MS2 performance while load increases. The best performing configuration was used to process a higher number of input files. Sample data with total file sizes increasing from 1.6 GB to 3.2 GB, 4.8 GB, 6.4 GB and 8 GB were used as input. Figure 8(a) shows that the ingestion process time increased almost linearly from 1.6 GB to 8 GB input files sizes in 36.8 seconds to 272 seconds (slope = 61.29). This increase of the duration in the processing stage was consistent as input files sizes kept increasing. Figure 8(b)

shows how the duration line of the processing stage compares to its linear line. From these experimental results we may conclude that the performance of MS2 is linear when total input file size increases.

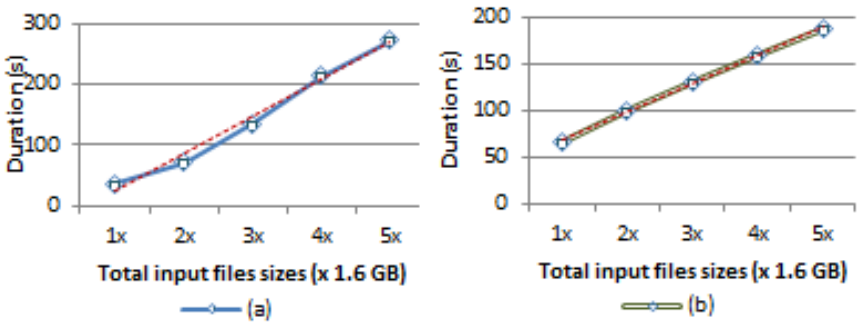

Figure 8 Performance linearity of (a) ingestion and (b) processing.

5 Related Works

At the time of writing, the number of publications specific to CDR preprocessing is still limited. Some works on CDR analysis techniques have been presented [21-23] but these techniques are designed for analytical processing after CDR has been preprocessed by mediation. Work on a middleware-based mediation system has been reported by Bouilett, *et al.* [4]. This system runs on IBM InfoSphere middleware and uses region-based parallelism to improve performance. Instead of using the same approach, MS2 uses record- and locality-based parallelism because region-based parallelism gives no benefit when CDRs are batched across different regions. Bouilett *et al.* reported that the resulted performance, i.e. 70,000 CDRs/second, was not adequate to meet business constraints at the time, which was 220,000 CDRs/second. Unfortunately, we cannot compare this performance to MS2 because the hardware configuration was not specified in their paper.

A pipeline-based parallel framework for mass file processing [24] can be used to process file-based CDRs; this is similar to the technique used in MS1. However, this technique is not optimal enough for CDR processing because it uses file-based decomposition instead of record-based decomposition. A scalable and distributed system of CDR stream analytics was proposed in [25]. The solution was optimized for streams where CDRs are continuously received in much smaller chunks. MapReduce is used to parallelize analytic processing after stream CDRs are previously merged and stored in a data warehouse. While there are additional computational costs for merging and storing in a data

warehouse, it will reduce performance when CDRs are already ingested in batches. Due to the lack of a performance evaluation in the paper, we cannot compare its performance to MS2.

Table 3 Performance of MS2 and related technologies.

System	Rec size (B)	Performance (rec/s)	MB /s	CPU (core @ 2.xGHz)	RAM (GB)
MS2	200	67,060	13.4	8	16
Kafka Producer (batch size=1)	200	50,000	10	16	32
Kafka Producer (batch size=50)	200	400,000	80	16	32
Kafka Consumer	200	22,000	4.4	16	32
iMR (MR Jobs)	7	13,000,000	91	80	960

Another class of related technologies comes from large-scale log processing, such as iMR [26], Kafka [27], and Flume [28]. iMR increases performance by moving analytics onto the log servers themselves. By transforming the data in place, this can increase locality and reduce volume of data crossing the network. But unlike batch-oriented workloads (as in the case of MS2), iMR takes as input small and continuous input streams and can take benefit in performance by only processing subsets of data and decreasing result fidelity (lossy processing). In many cases of CDR preprocessing, completeness is mandatory because unprocessed records may lead to missing transactions.

Both Kafka and Flume are focused on data loading such as log aggregation and message feeds. They lack parallelism model support for processing but are often combined with other platforms to deliver full real-time data analytics and processing [28,29]. We found that in our case, both Kafka and Flume can be used to load the CDRs to the processing nodes but as they are optimized for streams, this approach may introduce overhead for splitting larger sizes of batched CDRs to smaller chunks.

To give a view of how the efficiency of MS2 is compared with other related technologies, we also compared the performance based on resources used (i.e. CPU and RAM). We chose Kafka and iMR since the hardware configurations used in the experiments were published in the respective papers [27,28]. From Table 3, we can see that the Kafka producer is more efficient than MS2 with larger batch sizes but it performs no data processing, which is needed in case of MS2. The throughput of iMR on its test environment is higher than that of MS2 but the resources used are also higher than those used in the MS2 evaluation. If we adjust the resources based on ratio, MS2 has higher performance than iMR. MS2 performs better because it does not need the aggregation phase for continuous streams, which needs to be done in iMR.

6 Conclusion and Future Work

The need for CDR processing at telecommunication providers continues to grow and in the existing production system, it pushes the limits of what the system can deliver in terms of performance. To meet this need, a new CDR processing system was designed and implemented, called MS2 and based on a deep analysis of the existing system, which was presented in this paper. Also the results from an empirical evaluation of MS2 were presented, which demonstrate significant reductions in I/O overhead when using MS2, while delivering nearly 2 times improvement in throughput linear to the number of input CDRs. The performance of the CDR ingestion process was increased by using a multiplexing technique and sequential access. The performance of CDR preprocessing and the formatting process was increased by optimizing the MapReduce process and applying the correct tuning. Correct tuning here means that the system is configured to make optimal use of available CPU and memory resources.

However, the performance evaluated in this paper is based on a one-case scenario of CDR processing in a telecommunication mediation system. The performance result can possibly vary with different rules of CDR processing. Future research could evaluate the performance of the proposed system for several patterns of CDR processing rules. This evaluation can be also helpful for system improvement and further optimization.

Acknowledgement

We are very grateful to the anonymous reviewers for their helpful comments and suggestions.

References

[1] Jacobs, A., *The Pathologies of Big Data*, Communications of the ACM, **52**(8), pp. 36-44, 2009.

[2] McSherry, F., Isard, M. & Murray, D.G., *Scalability! But at what COST*, in 15th Workshop on Hot Topics in Operating Systems (HotOS XV), Kartause Ittingen, USENIX Association (2015), pp. 14, 2015.

[3] ITU-T, *X.690 Information Technology - ASN.1 Encoding Rules*, 1st ed., International Telecommunication Union, 2002.

[4] Bouillet, E., Kothari, R., Kumar, V., Mignet, L., Nathan, S., Ranganathan, A., Turaga, D.S., Udrea, O. & Verscheure, O., *Processing 6 billion CDRs/day: from research to production (experience report)*, in Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, Berlin, ACM (2012), pp. 264-267, 2012.

[5] Hohpe, G. & Bobby W., *Enterprise Integration Patterns*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[6] Bell, G., Gray, J. & Szalay, A., *Petascale computational systems*, Computer, **39**(1), pp. 110-112, 2006.
[7] Dean, J. & Sanjay, G., *MapReduce: simplified data processing on large clusters*, Communications of the ACM, **51**(1), pp. 107-113, 2008.
[8] Holmes, A., *Hadoop in Practice*, 1st ed., Manning Publications Co. Greenwich, CT, USA, 2012.
[9] Gray, J. & Prashant, S., *Rules of Thumb in Data Engineering*, in Data Engineering, IEEE (2000), pp. 3-10, 2000.
[10] Apache Hadoop, Apache Foundation, http://hadoop.apache.org, (1 July 2015).
[11] White, T., *Hadoop: The Definitive Guide*, 1st ed., O'Reilly Media Inc., 2012.
[12] Heger, D., *Hadoop Performance Tuning-A Pragmatic & Iterative Approach*, Computer Measurement Group Journal, **4**, pp. 97-113, 2013.
[13] Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., & Moon, B., *Parallel data processing with MapReduce: a survey*, AcM sIGMoD Record, **40**(4), pp. 11-20, 2012.
[14] Feng, X., Shen, J. & Fan, Y., *REST: An alternative to RPC for Web services architecture*, in Future Information Networks, IEEE (2009), pp. 7-10, 2009.
[15] Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S. & Saha, B., *Apache hadoop yarn: Yet Another Resource Negotiator*, in Proceedings of the 4th annual Symposium on Cloud Computing, Santa Clara, ACM (2013), pp. 5, 2013.
[16] Floratou, A., Patel, J.M., Shekita, E.J. & Tata, S., *Column-Oriented Storage Techniques for MapReduce*, Proceedings of the VLDB Endowment, **4**(7), 419-429, 2011.
[17] Gadkari, A., *Caching in the Distributed Environment*, Advances in Computer Science: an International Journal, **2**(1), pp. 9-16, 2013.
[18] Miner, D. & Adam S., *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*, 1st ed., O'Reilly Media Inc., 189-195, 2012.
[19] Joshi, S.B., *Apache Hadoop Performance-Tuning Methodologies And Best Practices*, in Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, Boston, ACM (2012), pp. 241-242, 2012.
[20] Chang, J., Lim, K.T., Byrne, J., Ramirez, L., & Ranganathan, P., *Workload Diversity and Dynamics in Big Data Analytics: Implications To System Designers*, in Proceedings of the 2nd Workshop on Architectures and Systems for Big Data, Portland, ACM (2012), pp. 21-26, 2012.

[21] Teng, W.G., & Chou, M.C., *Mining Communities of Acquainted Mobile Users on Call Detail Records*, in Proceedings of the 2007 ACM symposium on Applied computing, ACM (2007), pp. 957-958, 2007.

[22] Ding, L., Gu, J., Wang, Y. & Wu, J., *Analysis of Telephone Call Detail Records Based on Fuzzy Decision Tree*, Forensics in Telecommunications, Information, and Multimedia, **56**, pp. 301-311, 2011.

[23] Lin, Q. & Wan, Y., *Mobile Customer Clustering Based on Call Detail Records for Marketing Campaigns*, in Management and Service Science, IEEE (2009), pp. 1-4, 2009.

[24] Liu, T., Liu, Y., Wang, Q., Wang, X., Gao, F. & Qian, D., *Pipeline-Based Parallel Framework for Mass File Processing*, in Cloud and Service Computing (CSC), IEEE (2013), pp. 42-48, 2013.

[25] Chen, Q., & Hsu, M., *Scale out Parallel and Distributed CDR Stream Analytics*, Data Management in Grid and Peer-to-Peer Systems, Springer Berlin Heidelberg, **6265**, 124-136, 2010.

[26] Logothetis, D., Trezzo, C., Webb, K.C., & Yocum, K., *In-situ MapReduce for log processing*, in 2011 USENIX Annual Technical Conference, Portland, USENIX Association (2011), pp. 115, 2011.

[27] Kreps, J., Narkhede, N. & Rao, J., *Kafka: A Distributed Messaging System for Log Processing*, in Proceedings of the NetDB, ACM (2011), 2011.

[28] Liu, X., Iftikhar, N. & Xie, X., *Survey of Real-Time Processing Systems For Big Data*, in Proceedings of the 18th International Database Engineering & Applications Symposium, Porto, ACM (2014), pp. 356-361, 2014.

[29] Sumbaly, R., Kreps, J. & Shah, S., *The "Big Data" Ecosystem at LinkedIn*, in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, ACM (2013), pp. 1125-1134, 2013.