Replayable BigData for Multicore Processing and Statistically Rigid Sketching

Marat Zhanikeev*
* Department of Artificial Intelligence,
Computer Science and Systems Engineering,
Kyushu Institute of Technology
Kawazu 680-4, Iizuka, JAPAN 820-8502
Email: maratishe@gmail.com

***Abstract*— MapReduce/Hadoop is the de-facto default technology for processing BigData today. The processing itself is distributed (scale-out). However, recent research shows that processing throughput on singular multicore machines can easily exceed that of the distributed design. This paper proposes a new architecture that replays BigData on a single machine, processing the replayed stream in realtime on multicore. In order to make it work, objects in BigData are recorded with timestamps that makes it possible to replay BigData along the timeline. The ultimate goal of the architecture is a more statistically rigid processing environment which uses data streaming algorithms to produce sketches of targeted parts of BigData in realtime with minimal memory footprint.**

***Index Terms*— BigData replay, data streaming, statistical sketches, parallel processing, multicore processing, streaming algorithms, lockfree parallelization**

I. Introduction

Hadoop (HDFS) [20] and MapReduce are de-facto standards in BigData processing today. Although they are two separate technologies, they form a single package as far as Big Data *processing* – not just storage – is concerned. This paper will treat them as one package. Today, Hadoop and/or MapReduce lack popular alternatives [6]. Hadoop (HDFS) solves the practical problem of not being able to store Big Data on a single machine by distributing the storage over multiple nodes [3]. MapReduce is a framework on which one can run jobs that process the contents of the storage – also in a distributed manner – and generate statistical summaries. This paper will show that performance improvements are mostly found in the MapReduce part of the technology [19].

There are several fundamental problems with MapReduce. First, the *map* and *reduce* operators are restricted to key-value hashes (datatype, not hash function), which restricts usability. For example, MapReduce fails to accommodate the necessary datatypes or procedures required by most *data streaming* algorithms [8].

Secondly, MapReduce jobs create heterogeneous environments where jobs compete for the same resource with no guarantee of fairness [19]. Such a guarantee is difficult to achieve in distributed environments without incurring too much overhead.

Finally, MapReduce jobs, or HDFS for that matter, lack *time awareness*, while some algorithms might need to process in its time sequence and/or applying a time window. Most *data streaming* algorithms, for example, require that the incoming data stream is processed along a timeline.

The core proposal of this paper is to replace the HDFS/MapReduce pair with time-aware alternatives. Big Data is replayed along the timeline and all the processing jobs get time-ordered sequence of data items (objects, key-value pairs, etc.). Because of the replay, it is now more practical to process data on one machine. Note that traditionally HDFS/MapReduce sends jobs to remote nodes so that data can be processed locally (by/at other nodes). This does not cause a decrease in throughput because the processing itself is designed as a *lockfree multicore parallelization* which is shown to be extremely fast [2].

Having created a new framework, we can now use it for executing a wide range of statistically rigid data streaming algorithms [8]. Processing jobs run in parallel on multicore and can enjoy the complete freedom of datatypes in which to store *sketches* – statistical summaries of data. Note that traditional MapReduce operates only with the key-value datatype.

Note that data streaming is one of many possible practical applications of the proposed processing architecture. Data streaming is selected as a practical example in this paper simply because this particular application is currently at the final stage of software implementation which is why it is easier for this author to adopt the manual-like narrative that would allow the reader to duplicate the design.

II. Main Contributions of the Proposal

The generic essence of the proposal is as follows. With extremely high-volume replay – like that of BigData – one needs to partition input and process it via multiple sub-streams in parallel. There are two challenges here, both of which are resolved by this proposal.

Firstly, it is necessary to develop a brand new parallelization paradigm which would exploit the potential of a multicore architecture to its fullest. In other words, parallel processing on multicore should minimize overhead from synchronization across concurrent jobs. This proposal achieves this goal via a version of the *lockfree design* [2] which involves a special shared memory design but also an algorithm that minimizes cross-job (inter-process) communication overhead. The design

also involves a data construct in which data units are naturally listed in decreasing order of freshness (age) which allows for export and removal of old data units without cross-job (inter-process) synchronization.

Secondly, the process-while-replaying situation requires an original design that allows concurrent jobs not only to get access to the same stream of data, but also to process the stream efficiently. Efficiency here is a complex metric consisting of *space efficiency*, *processing speed* expressed as per-unit overhead, and most importantly, minimization of jitter across processing rates of concurrent jobs. The last metric is extremely important in practice because the entire system is forced to advance its timeline at the rate of its *slowest* job regardless or how fast other jobs are. This aspect is referred to as *heterogeneous jobs*. This paper proposes and implements an optimization problem that incorporates these issues. Statistical processing in each core is done using the *data streaming* paradigm [9].

The specific contributions of this paper are as follows. TABID (Time Aware BIg Data) is the name of the method and its software implementation. This abbreviation throughout this paper will refer to the proposal, its design and implementation. While Hadoop works with files on a filesystem, this paper proposes a *timeline data store* which is convenient for replay and can work with any datatype (key-value store, object store, etc.). The design of the data store itself is not key to this proposal but it serves as a proof that the proposal is valid and can be easily implemented in software. Since many jobs run concurrently at the replay node, this paper proposes a simple job packing heuristic which takes heterogeneity into consideration. With the packing heuristic, the proposal is a *natively multicore* technology according to the definition in [2].

Analysis of the proposal shows that replay-based architecture allows for more efficient use of resources, while MapReduce jobs have to read all files on all storage nodes. It is also shown that optimal packing can help maximize efficiency even for schedules with very many jobs running on commodity hardware with 8 cores.

III. Terminology

Hadoop and HDFS (HaDoop File System) are used interchangeably. In fact, for simplicity, Hadoop in this paper means *Hadoop and MapReduce*. Unit of action in Hadoop is *a job*. Note that these terms are used solely to provide a practical background to the proposal while the proposed method itself can be applied to other technologies as well.

Streaming algorithm and *data streaming* also denote the same technology. Here the unit of data is *a sketch* – a statistical summary of a bulk of data. *Sketches* function as jobs in the proposed method.

Data store is the same as *data storage*. Distributed parts of stores are called *shards* in Hadoop or *sub-stores* in this paper. Content itself is split in *records*, *key-value pairs*, *objects*, etc., while this paper prefers *items*. Items in this proposal

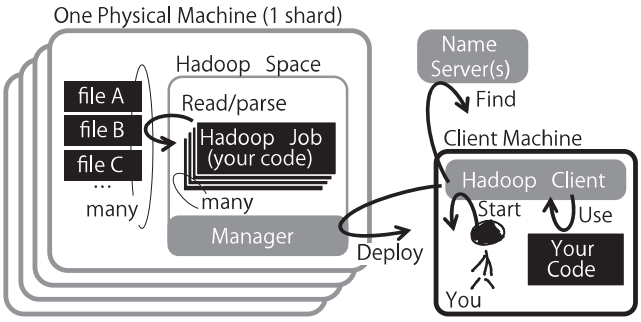

Figure 1: Architecture of a standard Hadoop/MapReduce system.

are assigned in time, which is why the store is referred to as *timeline store*.

The proposed method runs on *multicore* and needs *shared memory* so that the manager can communicate with *Sketches* running on cores. This paper employs a *ring buffer* to maintain a finite time window of items in shared memory continuously.

When estimating performance of the proposed method, *heterogeneity* is the main environmental parameter while *sketchbytes* (volume of data processed by all *Sketches*), and *overhead* (per data batch, per core) are the two quality metrics.

IV. Overview of Hadoop

Fig.1 shows the Hadoop architecture both in design components and key processes. Grey parts denote the system while sharp-black ones are for user files and code. The rest of this section will walk step-by-step through a standard MapReduce job.

The Code for the job has to be prepared in advance. Once ready, you can **start the job** by passing it to Hadoop (MapReduce, actually) client. The client will then **find** the shards scattered across the Hadoop cluster using Name Server and **deploy** your job to each.

Manager at each shard will run your code locally. The code can then **read** and **parse files** it finds locally, but more importantly perform the two key operations of MapReduce – **map** and **reduce**, where both operators are normally implemented by your code. The results are then sent back to the client machine, which, having collected data from all shards, will return the final result to the user.

Note that this is a simplistic view of the processes involved, but contains enough information to understand the key differences in the proposal presented further in this paper.

Let us consider one example that has difficulty running on Hadoop – the *counting frequent items* target popular in data streaming [11], among many others [8]. To execute it on Hadoop, we need to collect all items first using the *map* operator. We can then count all items using the *reduce* operator. However, at this point, we still need to keep all items in memory and, in fact, the data has to be transmitted over the network. Memory overflow is one potential problem in this scenario. But, in a larger picture, such a solution

violates the *space efficiency* objective commonly found in streaming algorithms [9]. Space efficiency can be achieved in data streaming mainly because items arrive in time order which allows for a statistically sound selection process and history management.

Another major problem with Hadoop is that it is incompatible with datatypes other than *key-value* hashes, where data streaming requires much greater flexibility. For example, Hadoop cannot work with *trees* and *graph* datatypes, many-to-many [1] and one-to-many [13] patterns, and others.

V. Related Work

Although Hadoop is unchallenged in practice today [6], the technology is known to be inefficient in several respects. Maximum achievable throughput for the HDFS system itself is found to be around 50Mbps [3]. HDFS is also found to be inefficient when content is split into a large number of small files [7], where the default block size used by HDFS is 64Mbytes and above.

Recently much attention in literature is paid to performance of Hadoop and its improvement. Statistics from a real commercially operated cluster are presented in [5], while workload modeling and synthesis are proposed in [4]. Performance improvement can be split into the following two groups. Parallel processing on multicore is proposed in [19] and requires a major re-write of the architecture. Research in [6] proposes using more of local processing in RAM rather than distribution processing over the network.

There is some literature on improvement of the HDFS technology without the MapReduce component. Research in [16] proposes creating a searchable version of HBase – a simple non-searchable spreadsheet datatype on top of HDFS. Another method in [15] creates an entirely new form of data storage on top of HDFS, arguably an alternative to HBase. Given that resources on substores are shared by all jobs running in parallel, optimizing heterogeneous access to HDFS is also a subject of interest [14].

Data streaming is a relatively new method but solves the same problem as MapReduce – high-volume realtime Big Data processing. Simply put, the main premise of data streaming is the ability to compress large volumes of data into small statistical summaries called *sketches*. The article in [8] is an excellent 100+ page introduction to the topic. Practical algorithms in literature are already applied to traffic analysis [9]. There is research on space efficiency designs and optimizations [10]. Temporal features of data streaming and sliding windows are considered in [12].

Well known practical data streaming targets are *frequent item discovery* [11], aggregation of one-to-many records such as found in quickly spreading viruses [13], and generic aggregation of many-to-many records [1]. However, the toolkit of data streaming is extremely flexible and can theoretically accommodate any practical target.

Data streaming is the main reason for the new design in this paper. Time awareness is implemented specifically with data streaming in mind. Coincidentally, the resulting architecture can benefit from running on multicore hardware.

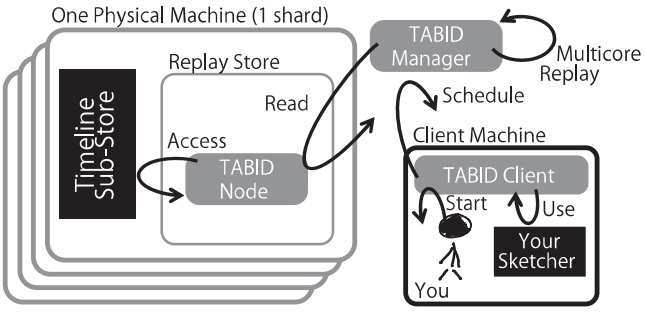

Figure 2: Architecture of the proposed Hadoop alternative – the TABID system.

Multicore is a special case of parallel processing [17]. It has already been applied to MapReduce in [19], which improves its performance but does not solve all the other problems listed above. Majority of existing multicore methods implement traditional scheduling-based parallelization [17] [18] [19], which requires intensive message exchange across jobs. This paper uses a special case of parallelization which features minimal overhead due to a lock-free design. More details are presented further in this paper.

VI. TABID: Time-Aware Big Data

Fig.2 presents architecture of the proposed TABID method. For simplicity, the same basic layout as in the earlier chapter on Hadoop is used. Also, mirroring the Hadoop piece above, this section will walk through a standard TABID process. APIs pertaining to some parts of the design are discussed in the next section.

Before the process, let us consider differences in *sharding*. Sharding is used in TABID as well, but the store is ordered along the timeline. Note that such a design may not need a Name Server because shards can be designed as *chains* with one shard pointing to the next. This discussion is left to further study and will be presented in future publications. Also note that, as the API will show further on, data items are not key-value pairs but are arbitrary strings, thus accommodating any data type like encoded (Base64, for example) JSON[21].

The process starts with defining your **Sketcher**. The format is JSON [21] and a fairly small size is expected. Unique features of Sketchers are discussed at the end of this section.

When you **start** your Sketch, TABID Client will **schedule** it with the TABID Manager. The schedule involves optimization where Sketches are packed in cores in realtime, as will be shown further in this paper. When a **replay** session starts, your Sketch will run on one of the cores and will have access to the timeline of items for processing. When the replay session is over, results are **returned** back to the client in a JSON datatype with arbitrary structure.

The following are the unique features/differences of TABID. First, there is *no code*. Instead, *Sketcher* specifies which

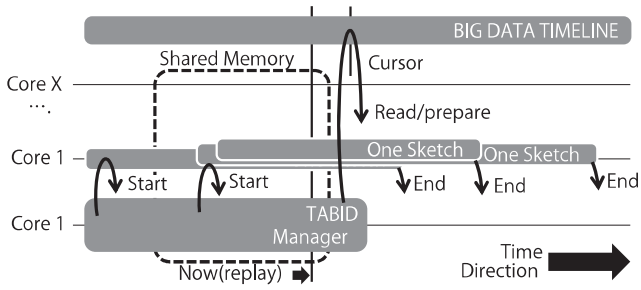

Figure 3: Big Data Timeline and sketches running in groups on multiple cores.

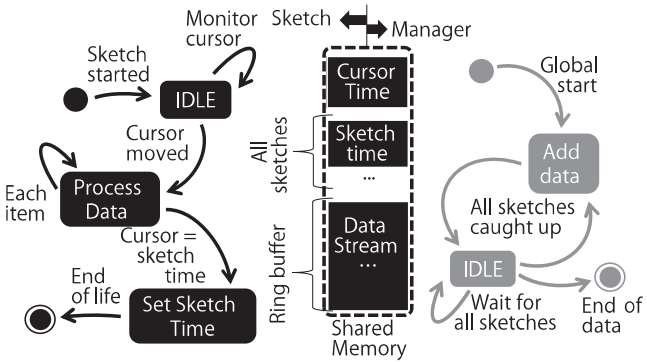

Figure 4: Shared memory design and logic followed by each Sketch *(left)* and TABID Manager *(right)*.

standard streaming algorithm is used and provides values for its configuration parameters. In case of a new streaming algorithm it should be added to the library first and then referred to via the Sketcher. Substore nodes in TABID are dumb storage devices and do not run the code, although it should not be difficult to create a version of TABID which would migrate to shards to replay data locally. Such a version with its performance evaluation will be studied in future publications.

VII. TABID: Streaming on Multicore

A method which packs n Sketches into m cores is a well-known *bin packing* problem. The method presented in this section performs one (blind) initial packing but then re-packs every time there is a change in state. Given that each Sketch has *start* and *end times* which can differ from those of the replayed Big Data, changes in state can be frequent. Given that all Sketches on all cores share the same time-ordered stream of items, the main objective of the heuristic is to minimize variance across processing *cursors* (current position in stream) across Sketches. Intuitively, if all Sketches with heavy-duty processing are packed into one core and all light-duty ones are packed into another, the replay will move at a greatly diminished speed. The proposed heuristic aspires to avoid such situations, while the design itself facilitates parallel processing without cores passing messages or competing for memory locks.

Fig.3 shows the design of the TABID Manager Node. The manager is running on one core and is in charge of starting and ending sketches as per its current schedule. Note that this point alone represents a much higher flexibility compared to MapReduce which has no scheduling component. Read/write access to the time-ordered stream, although remaining asynchronous, is collision-free by ensuring that writing and reading cursors never point to the same position in the stream. Sketchers read the data at the *now* cursor while the Manager is writing to a position further along the stream, thus creating a collision-safe buffer.

Fig.4 shows a simple shared memory design which is used by the Manager and all Sketches on all cores (reference C/C++ code in [22]). The figure also presents the logic for the two kinds of processes.

The Manager (right side) logic is as follows. Since it is helpful to avoid per-item signaling (via shared memory), data is read in batches within the physical limits of the *ring buffer* which in turn depends on the size of the shared memory. When a new batch has been read into the ring buffer, the global *now* cursor is updated, which allows for all Sketches to start processing new items. The Manager also monitors all *Sketch times* – cursors for individual Sketchers, and can change packing configuration based on collected statistics.

Each Sketcher (left side) follows the following logic. It waits for the *now* cursor to advance beyond its own cursor, which is stored independently for each Sketch in the shared memory. When it detects change, the Sketch processes all newly arrived items. Once its cursor reaches the global *now*, the Sketch returns to the idle state and starts polling for new change at a given time interval.

Note that ring buffer is accessed via a C/C++ library call [22] rather than directly by each party. This is a useful convention because it removes the need for each accessing party to maintain its current position in the ring buffer. The buffer only appears to be continuous while in reality it has finite size and has to wrap back to the head when its tail is reached.

VIII. TABID: APIs

Although far from an exhaustive list, the APIs in this section can fully describe TABID functionality. All APIs are executed over HTTP, even if they occur within the same physical machine.

To append a data item to the timeline store, one can send
POST tabidStorePut(timestamp, 'string'),
where *string* is completely freetype. It is common practice to send Base64-encoded JSON in this manner.

To schedule a Sketch, one sends the Sketcher JSON as
POST tabidSketchAdd(startTime, JSON),
which returns the *ID* of the newly created and scheduled Sketch. Its current status can be verified using
GET tabidSketchStatus(ID),
which returns status JSON.

IX. Analysis: Setup and Metrics

This section discusses parameters and metrics used for analysis. First, let us define the concept of a *configuration tuple*:

$$\langle v_{min}, v_{max}, a \rangle,$$

which specifies a distribution of values between v_{min} and v_{max} configured by a. The tuple can be used to define *heterogeneity*:

$$y = \begin{cases} norm_{v_{min}...v_{max}}\left(\exp^{-ax}\right)^{-1} & \forall\ x \in (1...100), a > 0, \\ v_{max}, \quad a = 0, \end{cases}$$

where a is the exponent and *norm* is the operator that normalizes the distribution of values and maps it onto the range between v_min and v_max. Note that the case of $a = 0$ is a special *homogenous* case. In analysis, values for a are specifically $0, 0.7, 0.3, 0.1, 0.05, 0.01$, where $a = 0$ is a homogenous distribution (horizontal line), a between 0.7 and 0.3 creates distributions with majority of large values, $a = 0.1$ is almost a linear trend and a between 0.05 and 0.01 outputs distributions where most values are small. The last two cases are commonly found in natural systems and can therefore be referred to as *realistic*.

Distributions apply to *Sketch lifespan* and *per-unit overhead* – the two practical metrics describing the performance of a TABID system.

The specific tuples used for overhead are $\langle 100, 10000, 0 \rangle$, $\langle 100, 10000, 0.7 \rangle$, $\langle 100, 10000, 0.3 \rangle$, $\langle 100, 10000, 0.1 \rangle$, $\langle 100, 10000, 0.05 \rangle$, and $\langle 100, 10000, 0.01 \rangle$, where unit of measure is *microsecond*. The specific lifespan tuples are $\langle 100, 2500, 0 \rangle$, $\langle 100, 2500, 0.7 \rangle$, $\langle 100, 2500, 0.3 \rangle$, $\langle 100, 2500, 0.1 \rangle$, $\langle 100, 2500, 0.05 \rangle$, and $\langle 100, 2500, 0.01 \rangle$, where unit of measure is *minutes*. It is therefore assumed that the example Big Data used for analysis is *2500m* long (in perfectly realtime replay). With the commonly found practical throughput of about 1Gbyte per minute, the Big Data in question is 2500Gbytes in size [3]. There can be between 10 and 1000 parallel Sketches and 8 cores (assuming commodity 8-core hardware).

The packing heuristic is defined as follows. C denotes a set of item counts for all sketches in all cores, one value per Sketch. M is a set of per-core item counts, one value per core. Optimization objective is then (var and max are operators):

$$minimize \quad var(C) + max(M).$$

For simplicity, values for C and M are normalized within a time window to avoid dealing with different units of measure in the two terms. In analysis, the problem is solved using Genetic Algorithm, for which the above objective serves as a *fitness function*. Detailed description of GA is omitted due to limited space.

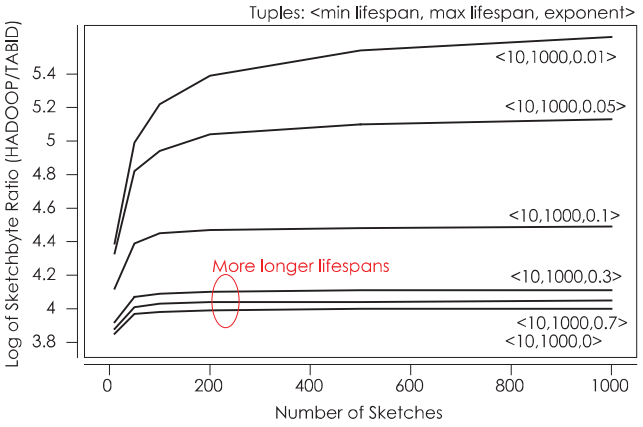

Figure 5: Performance of *sketchbytes* metric over a wide range of the number of sketches and several heterogeneity setups.

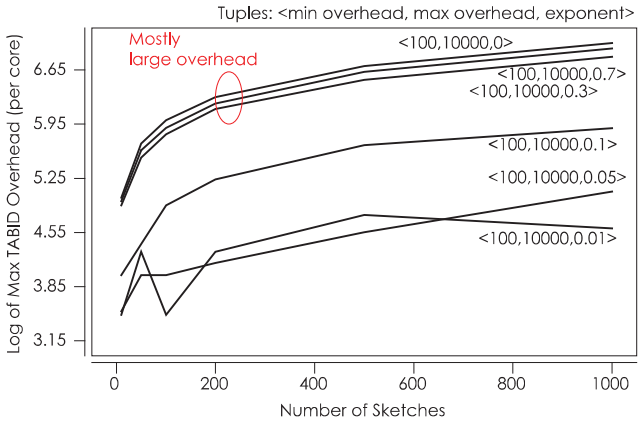

Figure 6: Performance of *max per-core overhead* metric over a wide range of the number of sketches and several heterogeneity setups.

X. Analysis: Results

This section presents results in two settings: performance in terms of *sketchbytes* versus *overhead*. The former is evaluated by the ratio of *sketchbytes* between traditional Hadoop jobs and the proposed TABID Sketches. The latter is a *worst-case* analysis only for TABID, which shows what item throughput can be expected given heterogeneity and number of sketches.

Fig.5 shows *sketchbytes* performance expressed as Hadoop / TABID ratio. In cases when heterogeneity $a \geq 0.1$, the ratio is around 1000. For smaller (more realistic) a, the ratio does not saturate but grows continuously up to 5-6 orders of magnitude (log scale). Note that this gap in *sketchbytes* is solely due to heterogeneity of Sketch lifespans. If to account for the fact that in TABID the stream is read/replayed only once, the gap widens further.

Fig.6 shows performance of the above packing heuristic. For $a \geq 0.1$ it is difficult to constrain continuous increase in maximum overhead (per core, not per Sketch). However,

for the realistic cases of $a = 0.05$ or even better for $a = 0.01$ there is a near-saturation trend, which means that performance is about the same for few Sketches as well as for very many Sketches. The worst case is about $5.4ms$ (see configuration above) per item (10ms is the configured max), which means that the slowest core can still process roughly 200 items per second per sketch. Note that the total throughput of the system is much greatly because of multiple Sketches and multiple cores.

XI. Conclusion

This paper proposed a fundamentally different approach to processing Big Data. The Big Data is replayed in a multicore environment on a single machine while many heterogeneous (processing) jobs run concurrently on separate cores, having access to the same data stream and producing statistical sketches on the output. The core of the proposal is a parallel processing design which almost entirely eliminates overhead from inter-process communication and specifically locking.

The proposed design is an alternative for the both parts of the HDFS/MapReduce technology. The HDFS part is replaced with a time-aware storage where each item is coupled with a timestamp. MapReduce is replaced with a time-aware replay on top of which multiple jobs can run multiple streaming algorithms in parallel. The new design is necessary to accommodate streaming algorithms whose complex and non-standard datatypes make it impossible to implement them as MapReduce jobs.

The proposed TABID design is fundamentally different from HDFS/MapReduce. In MapReduce, jobs are dispatched to remote machines to collect and bring back summaries. TABID replays data over the network as a single stream, from which each job (Sketch in TABID terminology) reads and processes items. Since all the Sketches in TABID run on the same machine, the next logical upgrade is to implement the system as a parallel process. In this paper, TABID runs on multicore.

In order to facilitate an efficient multicore parallelization, TABID incorporates an optimization problem which, using a simple heuristic, maps n Sketches into m cores in such a way as to minimize variance in processing positions (cursors) across Sketches. This implies that changes in state (entry and exit of Sketches, etc.) can cause a re-packing in order to improve efficiency of the system in its new state.

Analysis of TABID's performance showed that it is much more efficient than HDFS/MapReduce in terms of the volume of digested information as long as Sketches are heterogeneous in lifespan. Note that MapReduce jobs are also heterogeneous but MapReduce does not optimize resource use via scheduling. TABID was also found to be stable and independent of the number of Sketches, as long as majority of them would incur negligibly small overhead. Note that such a scenario is close to reality [5].

Some of the features of the proposal are left out of the paper due to size limitations. For example, time-aware storage makes it easy to discard oldest items. This is hard to do in HDFS because it has no time dimension. Instead, users have to timestamp files and remove old data manually, triggering large changes in HDFS. This and other additional features of the proposal will be considered in future publications.

References

[1] M.Zhanikeev, "A holistic community-based architecture for measuring end-to-end QoS at data centres", Inderscience International Journal of Computational Science and Engineering (IJCSE), 2014.

[2] M.Zhanikeev, "A lock-free shared memory design for high-throughput multicore packet traffic capture", International Journal of Network Management (IJNM), vol.24, pp.304–317, June 2014.

[3] K.Shvachko, "HDFS Scalability: the Limits to Growth", the Magazine of USENIX, vol.35, no.2, pp.6–16, 2012.

[4] Y.Chen, A.Ganapathi, R.Griffith, R.Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites", 19th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.390–399, July 2011.

[5] Z.Ren, X.Xu, J.Wan, W.Shi, M.Zhou, "Workload Characterization on a Production Hadoop Cluster: A Case Study on Taobao", IEEE International Symposium on Workload Characterization, pp.3–13, 2012.

[6] A.Rowstron, S.Narayanan, A.Donnelly, G.O'Shea, A.Douglas, "Nobody ever got fired for using Hadoop on a cluster", 1st International Workshop on Hot Topics in Cloud Data Processing, April 2012.

[7] Small File Problem in Hadoop (blog). [Online]. Available: http://amilaparanawithana.blogspot.jp/2012/06/small-file-problem-in-hadoop.html

[8] S.Muthukrishnan, "Data Streams: Algorithms and Applications", Foundations and Trends in Theoretical Computer Science, vol.1, no.2, pp.117–236, 2005.

[9] M.Sung, A.Kumar, L.Li, J.Wang, J.Xu, "Scalable and Efficient Data Streaming Algorithms for Detecting Common Content in Internet Traffic", ICDE Workshop, 2006.

[10] Z.Bar-Yossef, R.Kumar, and D.Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs", 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2002.

[11] M.Charikar, K.Chen, and M.Farach-Colton, "Finding frequent items in data streams", 29th International Colloquium on Automata, Languages, and Programming, 2002.

[12] M.Datar, A.Gionis, P.Indyk, and R.Motwani, "Maintaining stream statistics over sliding windows", SIAM Journal on Computing, vol.31(6), pp.1794–1813, 2002.

[13] S.Venkataraman, D.Song, P.Gibbons, A.Blum, "New Streaming Algorithms for Fast Detection of Superspreaders", Distributed System Security Symposium (NDSS), 2005.

[14] A.Rasooli, D.Down, "COSHH: A Classification and Optimization based Scheduler for Heterogeneous Hadoop Systems", Technical report of McMaster University, Canada, 2013.

[15] S.Das, Y.Sismanis, K.Beyer, R.Gemulla, P.Haas, J.McPherson, "Ricardo: Integrating R and Hadoop", SIGMOD, pp.987–999, June 2010.

[16] X.Gao, V.Nachankar, J.Qiu, "Experimenting with Lucene Index on HBase in an HPC Environment", 1st Annual Workshop on High Performance Computing Meets Databases (HPCDB), pp.25–28, 2012.

[17] M.Aldinucci, M.Torquati, M.Meneghin, "FastFlow: Efficient Parallel Streaming Applications on Multi-core", Technical Report no. TR-09-12, Universita di Pisa, September 2009.

[18] R.Brightwell, "Workshop on Managed Many-Core Systems", 1st Workshop on Managed Many-Core Systems, 2008.

[19] R.Chen, H.Chen, B.Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling", 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp.523–534, 2010.

[20] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[21] JSON format. [Online]. Available: www.json.org

[22] MCoreMemory project page. [Online]. Available: https://github.com/maratishe/mcorememory