Parallel Bulk Insertion for Large-scale Analytics Applications

Antonio Barbuzzi
Politecnico di Bari
a.barbuzzi@poliba.it

Pietro Michiardi
Eurecom
pietro.michiardi@eurecom.fr

Ernst Biersack
Eurecom
ernst.biersack@eurecom.fr

Gennaro Boggia
Politecnico di Bari
g.boggia@poliba.it

ABSTRACT

Modern data analytics applications, e.g. Internet-scale indexing, system trace analysis, recommender engines to name a few, operate on massive amounts of data and call for a parallel approach to data processing. In this work, we focus on the popular MapReduce framework to carry out such tasks and identify bulk data insert operations as a critical preliminary step to achieve reduced processing times, especially when new data is generated and processed at regular time intervals.

We present a *parallel approach to bulk data insertion* in a system that use horizontally range partitioned data and evaluate several variants to insertion operations, including legacy approaches. Our method exploits the parallel processing framework itself to insert data into the system, which is stored in a semi-structured format. Our results indicate that a parallel approach to bulk insertion can substantially reduce the recurrent costs of insertion of new data into the system.

1. INTRODUCTION

The MapReduce (MR) framework [8], designed by Google and democratized by the Hadoop project [3], has paved the way for massive-scale data analytics applications. One reason for the success of the MR framework is that it relies on a distributed system to store and process data that scales well, tolerate faults and that can be deployed on a cluster of "commodity" hardware. Current best-practice deployments [9] suggest to couple the MR framework with a distributed database such as BigTable [5] or HBase [2], with the goal of avoiding to read unnecessary data.

In this work, we address the fundamental problem of efficient bulk data insert in such a distributed framework. Bulk insertion is crucial since it is a recurrent operation: data is collected over a period of time (e.g. a day), aggregated and then bulk inserted into a parallel processing framework. Although some works [12, 4, 7] showed that bulk insertion accounts for a substantial fraction of the total time required to obtain processed results, this problem has been framed for the first time in [13], for horizontally range-partitioned distributed databases.

The general principle underlying many data management systems that support parallel processing frameworks is that of horizontally partition data over a cluster of shared-nothing machines. There are two kind of horizontal partitioning techniques, *hash* based and *range* based partitioning: the former scheme uses the hash of the key to decide each key-value destination, while in the second scheme each partition corresponds to a range of sorted primary-key values. Examples of such systems are Dynamo [10], Cassandra [11], BigTable [5], PNUTS [6], and HBase [2].

In this work, we focus on systems that use range partitioning. In this case, bulk insertion can be problematic: when a key range is hot, the machine assigned to it can be overloaded. For example, consider data to be range partitioned by a timestamp value. As bulk insertion takes place at regular time intervals, new records will fall in a small subset of the timestamp range. This creates hot-spots in those machines that are in charge of such range: overall system performance decreases and parallelism is lost. Instead, when hash partitioning is used, a natural load balancing avoids hot-spots. However, many analytic tasks require access to a range of the key-space, hence this approach is not suitable.

Motivated by our current efforts toward processing a large amount of network traces for troubleshooting and traffic analysis using the MR framework and HBase, we study efficient techniques for bulk insertion. Instead of enhancing the architecture of HBase similarly to what has been done for PNUTS [6] in [13], we propose a *parallel approach to bulk insertion* that uses the MR framework itself to write data in HBase. We show that for many analytical applications that require range partitioned data, our method is very efficient: it involves all machines of the cluster in the insert operation, while no data movement across machines is necessary. Through a series of experiments, we show the benefits of our approach and compare it to current best-practices in bulk insertion. While in the rest of the paper we focus on HBase, such approach could be easily adapted to other range based partitioned systems.

LADIS 2010 Zurich, Switzerland

The remainder of the paper is organized as follows: in Sec. 2 we discuss the bulk insertion problem in details, and outline the method described in [13]. In Sec. 3 we describe our technique, which is evaluated in Sec. 4, and conclude in Sec. 5.

2. SYSTEM OVERVIEW

In this Section, we outline the system we use in this work and describe in details the problem raised by bulk insert operations. Here we gloss over several details of each building blocks of the system, and focus on HBase, which is mostly affected by insert operations.

In this work we the Hadoop framework [3]: the HDFS distributed file system, Hadoop MR and HBase, the distributed database.

HBase is a distributed column-oriented database built on the top of HDFS. Its main storage containers are the `Tables`, conceptually similar to the database equivalents. Data is stored and retrieved by key, and table rows are sorted by row key. We now overview the most important components of HBase. In HBase, tables are automatically partitioned horizontally into `Regions`; each region includes a non overlapping and contiguous subset of the tables defined by their first and last keys. Whenever a region grows over a threshold, it is split in two new regions, of roughly the same size. Regions are assigned to `region servers (RSs)`, the nodes that are responsible for read, write and split operations. The `HMaster` is responsible of coordinating and orchestrating all the nodes of the cluster; it accounts for assigning regions to RSs, bootstrapping and failure recovery. Regions are assigned in a pseudo-random order to the RSs. Data is not moved around aggressively in order to achieve load balancing. HBase uses `Zookeeper` [1] to maintain the cluster state. `HFiles`, the equivalent of BigTable *SSTable*, are the actual row storage structure where data is written. They contain an ordered sequence of key/values and some indexes, to speed up data lookup. Using the Zookeeper cluster, fresh clients obtain a table holding the list of all regions and their start keys, from which they can learn the location of each row in the table.

When a distributed database system, such as HBase, stores range partitioned data, it is important to avoid overloading individual cluster nodes when a particular portion of the range is hot.

In the following, we illustrate the problem of bulk insert with the simplifying assumption of an empty database (Fig. 1). Suppose data is indexed using a (unique) timestamp as key: data is partitioned according to the timestamp value. Furthermore we assume data sorted according to the partition key and inserted chronologically. The key space is represented using a line, divided in days. The nodes of the cluster are represented as circles, ranging from A to G. The figure is split in four parts, representing four insertion phases.

Initially the table is empty. As the first key value is inserted in the table, the whole key space is assigned to a single region, hosted by a single RS, namely A. All subsequent insertions are supported by node A (see Fig. Fig. 1.1).

As soon as the region size reaches a threshold, a *split* operation is required: the partition is divided in two new partitions (see Fig. 1.2). One partition will hold half of the data written so far, the other partition will hold the other half. In general, the HBase system will assign one of the two new partitions to a different machine to achieve load balancing. As a consequence, a *move* operation, *i.e.* moving data from the original node to the new node, is also required. In the example, the RS A hosting the first partition takes care of range *Monday 0-6 am*, RS B of all the remaining key range (starting from Monday 6 am). In Fig. 1.3 the process continues. Since data is inserted in order according to the timestamp, all the following insertions affect node B only. In Fig. 1.4, when the insertion point reaches *Monday 18 pm*, the region hosted by node B fills up and is split in two equally sized parts. The first half covers keys from Monday 6 am to Monday 12 pm; the second half covers the whole remaining key space and takes half of the previous split. The last region is assigned to node C, that will will support the whole insert load. Indeed, the range affected by new data is the one assigned to the new node. This phenomenon propagates as new splits and move operations take place, when the partition size is not sufficient to accommodate new data. Note that when the assumption of an empty database is dropped, this problem aggravates: split and move operations are more frequent and involve more cluster nodes.

In the above example we considered only one possible scenario for bulk insert. As illustrated in [13], there are several other cases that can be studied. In general, when no split operations are required (because the new data to insert in the database is small), bulk insert is not critical. In this work we consider a *dense* scenario, in which bulk insert involves a large amount of data. Note also that data can be inserted in a random or in a sorted order (based on range keys). We focus on the latter case, which is often encountered in practice. For example log files, network traces, e-commerce transactions all have a natural order based on the timestamp of an event.

An architecture for efficient bulk insert and its application to PNUTS has been proposed in [13]. The main idea behind the design of such a system, which requires auxiliary components to complement the legacy PNUTS data store, is the following. Bulk insert is prepared through a *planning phase*, in which a central component minimizes the maximum makespan on each machine, computed as the sum of the costs payed to move and insert data in the system. In practice, partitions are pre-computed so that both the insert and the move load is evenly shared among all the nodes of the cluster and the insertion time is minimized. Once the planning phase is completed, a number of *staging servers* proceed in parallel to insert new data into the machines in charge of each partition.

In this work we show that it is possible to achieve efficient bulk insert *without* having to modify the underlying data management system and use additional nodes, that can add significant complexity and cost to the system. Bulk insert is executed using the MR framework: all nodes of the cluster are equally involved in insert operations, and no optimization phase is required. Our method is particularly adapted for dense bulk inserts, and when data is generated and read in sorted order.

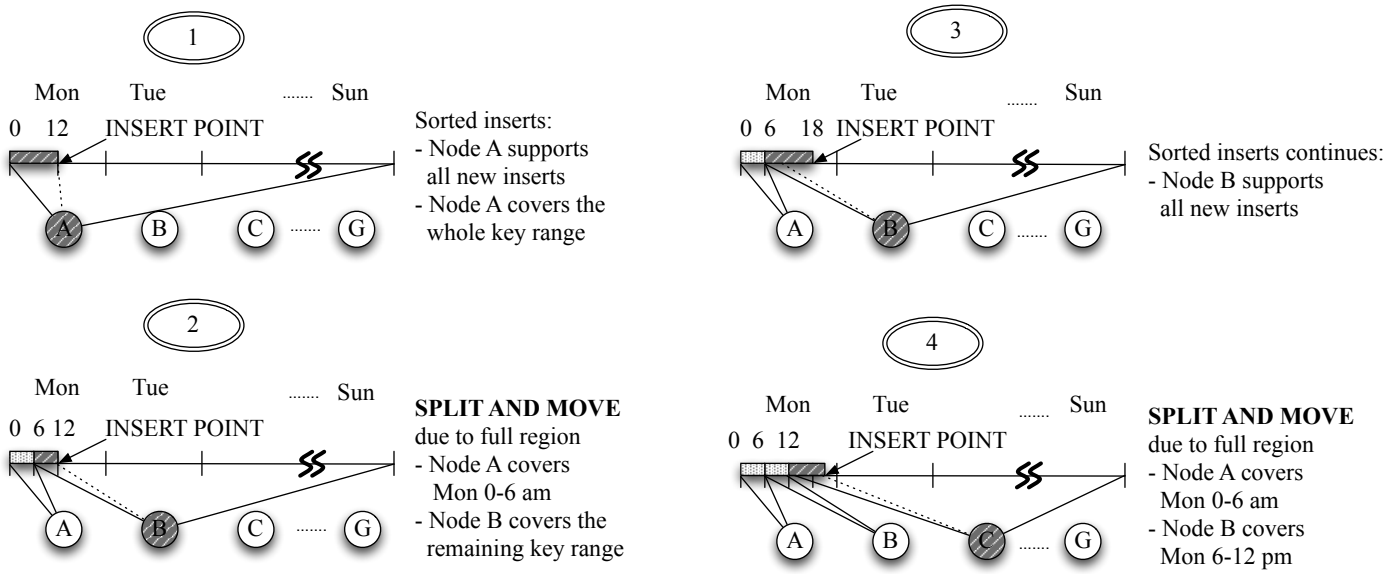

Figure 1: Example of sorted insertion of timestamp ordered data in a range partitioned distributed database. All the insertions are supported by a single node, which covers the most recent key range.

Before proceeding any further, we now outline the data flow that characterizes bulk inserts in our system, which is illustrated in Fig. 2. A data source collects raw data that is aggregated and written on the HDFS. HDFS is particularly suited for streaming data access, with typical patterns consisting of a single write operation and multiple read operations. Read/write latency is sacrificed in favor of high throughput. Raw data is split into equally sized blocks stored at different locations in the cluster. Data blocks can be replicated by HDFS to cope with failures. Note that data uploaded to HDFS does not need to be structured: hence, no pre-processing operations are required during this phase. However, if the input data is not splittable, e.g. due to the absence of record boundaries, the data source is required to introduce sync points to help MR jobs perform parsing operations.

Once raw data is on HDFS, bulk inserts in HBase can take place. As hinted above, we use the MR framework to perform such operation. Each mapper reads data from the local (HDFS) node and parses and analyzes it to create an assignment of data ranges to reducers such that each reducer will be involved in inserting roughly the same amount of data. Finally, reducers executed by each node of the cluster collect data from the mappers and write it in parallel to HBase.

3. PARALLEL BULK INSERTION

We now describe how the MR framework can be stretched to go beyond analytic tasks: in what follows we describe a *parallel approach to bulk data insert*. Despite its conceptual simplicity, the design of appropriate MR jobs to perform such tasks is far from being trivial. Recall that the HBase system relies on RS that store a subset, termed *region*, of the row keys of data tables. Each RS is responsible for one or multiple *column families*. Note also that tables are sorted based on row keys and column families are sorted based on column keys.

Our approach directly produces, sorts and positions HFiles, which are the materialization of a column family. A bulk insert job consists of both a *Map* and a *Reduce* phase. In the Map phase, data is read in parallel by all the mappers from HDFS using the native HDFS client interface; in the Reduce phase, HFiles are written in parallel by all the reducers back to HDFS for HBase to consume them. HFiles need to be appropriately sorted and placed to achieve load balancing across region servers in terms of number of row keys to store.

Between the Map and the Reduce tasks, we proceed with an intermediate sorting phase. In practice, the goal is to assign reducers an ordered range of mapper output keys to allow the Reduce phase to produce "well-formed" HFiles.

The default partitioner used in the MR framework applies a hash function to each key to determine a mapping between the (sorted) output key space of a mapper and the reducers. Each reducer is responsible for a set of roughly the same size of mapper output keys. However, note that the sort phase does not produce a globally sorted output, because each mapper only deals with a subset of the global key space. Therefore, reducers may operate on interleaved key ranges, e.g. a reducer could deal with keys $(1, 2, 5)$ and another with keys $(3, 4, 6)$.

Hence, our data insertion MR job uses an order preserving partitioner, which maintains a total order among the whole mapper output key space and avoids reducers to operate on interleaved key ranges. However, such a partitioner introduces an additional problem: the key space cannot be easily split among multiple reducers. Indeed the distribution of the keys in the mapper key space is rarely uniform and some reducers may end up being assigned a key range containing a small number of keys while other reducers may be overloaded with a densely populated key range. We solve this problem with a preliminary *sampling phase* on the global

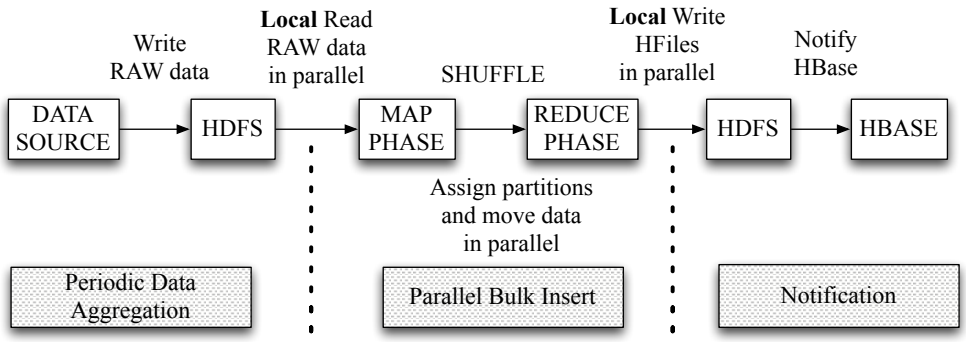

Figure 2: Data flow

key space, which is executed before the data insertion MR job by the client in charge of the data insertion. Note that the sampling phase is lightweight and fast, as it does not require a full scan of the dataset.

The MR framework comes with a set of three samplers: the "SplitSampler" takes a fixed number of keys to sample from the beginning of a fixed number of HDFS chunks; the "RandomSampler" is similar to the SplitSampler, but chooses the keys to sample uniformly at random; finally, the "IntervalSampler" selects keys at regular intervals through the HDFS chunks. The choice of a particular sampler depends on the nature of the dataset to be analyzed and, if needed, a new sampler can be easily coded. In this work, we use the "IntervalSampler", which is a good choice when the key space to analyze is sorted[1]. Note that the samplers provided with the MR framework can only operate on the input key space of the mappers. However, our requirement is to balance the load of reducers, which would require sampling the output key space of mappers. Indeed, the partitioning (*i.e.*, the "shuffle" phase) is performed on the mapper output keys. In our work, we set the input/output key space of a mapper to be equal, thus the sampling phase can be done without modifications to the MR framework.

4. EVALUATION

We now proceed with an evaluation of our approach to parallel bulk insert. We compare the performance achieved by our method to that of two baseline approaches that are currently used in real deployments.

Inserting data directly in the HDFS constitutes a typical approach to bulk data insertion. In this case, a single source writes data to the cluster. With this method, the MR framework requires full scans of the entire dataset to parse data, which constitutes a recurring computational overhead for all nodes involved in Map operations. We label this method *HDFS*.

An alternative approach is to use HBase as the sink of a data stream. In this case, the source needs to produce semi-structured data which introduce computational overhead at the client node. However, once data is inserted in HBase, MR jobs are greatly simplified since data analysis tasks neither involve parsing nor a complete scan of the dataset. This approach is particularly effective when the data to insert is not too big or if the data is produced in real time and at small rate. Note that in this case, the speed can be improved by using multiple threads to write to HBase. We label this method *HBase*.

Additionally, we study the performance of an alternative parallel bulk insert method based on the MR framework. This method relies on the native HBase interface instead of directly writing HFiles as described in Sec. 3. In this case, the legacy HBase system takes care of split and move operations. First data is uploaded to HDFS, as described above. Subsequently, a *data insertion MR job*, made by a *single Map phase*, takes care of writing data in HBase. Data is read from HDFS and written to HBase using the respective native interfaces. Read and write operations are performed in parallel by all the nodes of the cluster. This simple approach to parallel data insertion heavily relies on HBase to appropriately organize and sort data. We label this method *Parallel HBase*.

Tab. 1 includes the 2 default methods HDFS, and HBASE, parallel HBase and our method discussed in Sec. 3 and labeled *Parallel HFile*.

For our experiments we used a cluster of 34 nodes: a single node features a dual core processor, 4 GB of RAM and 250 GB of disk. All servers reside in the same network and are connected using a Gigabit switch. Four machines are permanently assigned to individually host the necessary auxiliary services that control our cluster: a filesystem metadata server (the NameNode) required by HDFS, Zookeeper required by HBase, the HBase metadata server, and the master controller of the MR framework (the JobTracker). The other machines are used to instantiate the distributed components of HDFS (the DataNode), HBase (the RegionServer) and MR (the TaskTracker), and we vary their number across the experiments, from 10 to 30. In this work, we used the Cloudera CDH3 distribution of Hadoop MR and HDFS, and we patched the latest release of HBase following the best practices indicated in [2].

The dataset used to perform the bulk data insert experiments originally reside on a separate client machine and consists of 11 GB of network traces collected from the internal LAN of our laboratory. The dataset is formatted accord-

[1]For sake of completeness, we used the dataset file position indicator as the input key to mappers and reducers.

	Default		Parallel	
	HDFS	HBASE	HBase	HFile
10 nodes	17 min	286 min	212+17 min	25+17 min
20 nodes	17 min	264 min	114+17 min	14+17 min
30 nodes	17 min	239 min	89+17 min	10+17 min

Table 1: Execution time of different insertion strategies

ing to the PCap specification[2], which is a typical example of an un-splittable data format. In particular, our dataset includes a global header, followed by the records of captured packets. The record size is not fixed and each record contains a field *incl_len* indicating the record size. The dataset contains no sync points, hence a parsing phase executed at the client is required. We used the default, 64 MB, split size of HDFS: we thus have a total of 174 splits.

Tab. 1 summarizes the results we obtained with the various bulk data insertion approaches we discussed. The values reported in Tab. 1 are averaged over 5 experiment runs, and the variance is in the order of seconds.

For the default approaches, uploading data directly to HDFS allows, as expected, to obtain the best write throughput values. However, the dataset is stored in a form that requires the data analysis MR jobs to do the heavy lifting of scanning (possibly multiple times) the whole dataset. Instead, writing directly to HBase allows analytics operations to avoid full scans reading unnecessary data. This comes at the cost of a dramatic increase in the insertion times, which is order of magnitude larger for the second default approach. For the parallel approaches, we observe that Method 1, which uses a MR job to write in HBase, reduces by roughly 30% the data upload time with respect to the HBase default method. This result can be greatly improved with the approach described in Sec. 3, which bypasses the HBase client interface. Data upload times are reduced by one order of magnitude with respect to the HBase default method. Moreover, parallel methods scale better with the number of hosts belonging to the cluster, as shown in the table.

5. CONCLUSION

In this work, we examined a variety of approaches to bulk data insertion, a recurrent operation that consists in importing a dataset into a cluster to be processed using the MapReduce framework. We showed that a parallel approach to data insertion can consistently speed up this critical preliminary step, that has been reported to account for a large fraction of the total time to obtain processed results in prior works on data-intensive analytics applications. Our most advanced method required an elaborate use of the MapReduce framework itself to carry out insertion operations. As a result, data was stored in a semi-structured manner, which can substantially benefit analytic operations by avoiding reading unnecessary data.

Our research agenda includes an extension of our method to accommodate complex scenarios with non-empty database and with multiple input sources.

[2] `wiki.wireshark.org/Development/LibpcapFileFormat`

6. REFERENCES

[1] Apache ZooKeeper: `hadoop.apache.org/zookeeper`.

[2] Hadoop HBase: `hadoop.apache.org/hbase`.

[3] Hadoop project: `hadoop.apache.org`.

[4] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB'09: Proceedings of the 2009 VLDB Endowment*, Lyon, France, Aug. 2009.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, Nov. 2006. USENIX Association.

[6] Community Systems Group. Community systems research at yahoo! *SIGMOD Rec.*, 36(3):47–54, Sept. 2007.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10: Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, New York, NY, USA, June 2010. ACM.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool, Jan. 2010.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220, Sept. 2007.

[11] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. In *LADIS '09: Proceedings of the 3nd Workshop on Large-Scale Distributed Systems and Middleware*, Big Sky Resort, Big Sky, MT, Oct. 2009. ACM.

[12] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 165–178, New York, NY, USA, 2009. ACM.

[13] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 765–778, New York, NY, USA, Aug. 2008. ACM.