Local Aggregation with Modified B+ tree in Map Reduce Data Processing

Ohnmar Aung
University of Computer Studies, Yangon
ohnmaraung2008@gmail.com

Abstract

MapReduce is well-applied in high performance computing for large scale data processing. However, as long as the clusters grow, handling with huge amount of intermediate data produced in the shuffle and reduce phases (middle step of Map Reduce) have impacts heavily upon the performance. With local aggregation (either combiners or in-mapper), shuffling large amounts of data can be reduced which alleviates the reduce straggler problem. The proposed modified B+ tree based indexing algorithm is applied to reduce intermediate data amount for output retrieval fast as well as scalable data storage.

1. Introduction

More businesses are becoming aware of the relevance of the data in which they are able to gather: from social websites to log files, and also there is a lot of hidden information ready to be processed and mined. It was neither difficult to work with large amounts of data nor hard drive capacity. Access speed is main point. Nowadays, it is much easier for companies to become global target a larger number of clients and consequently deal with more data [6].

Hadoop is a distributed file system and is coupled with parallel processing framework called, Map Reduce. It is widely used for large scale data processing and has benefited from its simplified two-step processes: map and reduce phase. Both of these two stages are separated by a barrier called shuffling, to ensure that all relevant input data is available to the reducer function before the latter one proceeds [1]. However, at large data volumes (input), tens of terabytes of data (intermediate data produced between the map and the reduce phase) have to be transported across the cluster machines and the associated overheads can become significant factor in the job's overall run-time. Although, a simple word count job can rarely result in a lot of intermediate data to transfer over the network, other jobs may produce a lot of intermediate data such as sorting a terabyte of data where the output of the Map Reduce job is a new set of data equal to the size of the data when started with [2]. Since the intermediate and output data sizes are quite large, various ways like custom combiner, reducer and even secondary sorting are considered to shrink those because the result-combining step adds a large overhead to the total run time and frequently can cause the network bottleneck. Therefore, the middle step (shuffle and sort) is the most important part of the framework, where the magic happens. It is said that well understanding of this work flows allows the optimization both the framework and the execution time of Map Reduce jobs [8].

The model allows developers to write massively parallel applications without much effort and is becoming an essential tool in the software stack of many companies that need to deal with large datasets. And even though its interface is simple, it has proved to be powerful enough to solve a wide-range of real-world problems: from web indexing to image analysis up to clustering algorithms. And also careful attention to partitioning data, scheduling, handling machine failures and inter machine communication can heavily impact on the system's performance like scalability, integrity and high throughput [3].

The proposed B+ tree-based indexing algorithm is applied in the shuffle and sort phase in order to aid in the reduction of the amount of intermediate data. Section 2 is discussed about related work. The background theory applied is described in Section 3. The overview of the current and new approach is explained in Section 4 and Section 5. As for Section 6, complexity from the view of theory is intended and all of the discussion is concluded in Section 7.

2. Related Work

Yahoo-Sailfish introduced L-files as an abstraction, implemented as an extension of the distributed file system for supporting network-wide data aggregation. Its L files made batching of data written by multiple writers and then transported intermediate data (specially, to transfer output of map tasks to relevant reduce tasks). But, there was a blocking whenever the output from one step has to be materialized by writing to disk-based storage before it can be consumed by a later step. This may lead to sometimes traffic jam if one of the steps (especially earliest stage) in the processing takes longer time than as usual [9].

Map Reduce model used a barrier between the Map and Reduce stages for simplifying in both of programming and implementation. But, in many situations, this barrier hurt performance because it was overly restrictive. The author and his colleges developed method to break the barrier in Map Reduce in a way that improved efficiency [1]. However, this looks like using zero reducer which is not suitable to every applications based on Map Reduce applications.

One distributed network file system like Wofs[11] that split a file into many small objects, stored these objects in remote file servers, and used a special B+ tree to manage the metadata of these objects. Besides, it used the object-range locking policy to avoid data incoherence and improve performance.

Gongye Zhou and his companions [12] proposed a B+ Tree Management Method of Object Attributes for Object-based Storage. That controled storage attributes with two-level B+ tree structure: one for attributes index and another for object index.

None of the latter two systems (based on B+ tree) considered limiting the order and height of B+ Tree. This becomes critical issue for system performance.

The proposed system deeply takes into account “the order and height” of the tree for collecting intermediate data using in-mapper combining function before moving to reducer. This can reduce the amount of transferred data that need to be shuffled across the network. Besides, the system can compromise the complexity due to frequent insertion and deletion not being higher than original B+ tree.

3. Background Theory

3.1 Map Reduce Data Flow

In Map Reduce, the map function emits each word plus an associated count of occurrences whereas the reduce function sums together all counts emitted for a particular word [4]. The model also makes the guarantee that the input to the every reducer is sorted by key. The process by which the system performs the sort-and transfers the map outputs to the reducers as inputs- is known as the shuffle where a large amount of intermediate data can be produced, which is the heart of Map Reduce [10].

The underlying mechanism used for handling intermediate data in a Map-Reduce computation is via a parallel merge-sort. The cost of handling intermediate data depends on (1) inter-node connectivity within the cluster and (2) the rate at which data can be read from (as well as written to) the disk subsystems on individual nodes. The effective disk transfer rate is highly dependent on the number of seeks as well as the amount of data read per disk seek. Unless careful attention is paid to the seek overheads involved in handling the intermediate data, cluster throughput will degrade [9].

3.2 B +Tree

Many tree-based algorithms are used to store data; however, they cannot handle the entire tree status of balancing after some operations like insertion and deletion. Those might store for fast efficient insertion well, but bad ending in deletion. Consequently, maintaining system’s balance after deletion becomes a major problem in today’s tree-based storage area [13].

Other balanced trees such as AVL trees and Red-Black Trees use the height of the sub-trees for balancing whereas WBT (Weighted Balanced Trees) is based on the size of the sub-trees below each node. Weighted balanced trees are well suited for organizing data orderly associated with their size. But, frequent insertion and deletion makes the tree order difficult and waste time [14].

A simple B+ tree consists of one or more blocks of data, called nodes, linked together by pointers. Like many tree-based approach, it has three types of basic nodes: root, internal nodes and leaf. Internal nodes which are used as an index nodes that point to other nodes (child nodes) in the tree. Leaf nodes, is also called data nodes which maintains data page as well as pointer to neighboring nodes via doubly linked list. Data searching in the tree always starts at the root node and moves downwards until it reaches a leaf node. Both internal and leaf nodes contain key values that are used to guide the search for entries in the index. It is also a balanced tree due to the fact that every path from the root node to a leaf node is the same length. Major emphasis on B+ tree is to consider the order of the tree that make how large it can be [12].

4. Handling Intermediate Data

In Map Reduce data processing, local aggregation of intermediate result is one of the keys to be algorithms efficient. With local aggregation (either combiners or in-mapper combining), it reduces the number of values associated with frequently-occurring terms, which alleviates the reduce straggler problem [5]. In this section, analysis of using combiners vs in-mapper, stripe vs pair, merge sort is discussed and proposed B+ tree based algorithm is proved as a more efficient one over the existing approaches.

4.1 Using Combiners or In-Mapper

Combiner does local aggregation of key/values produced by mapper before or during shuffle and sort state of Map Reduce processing in order to significantly reduce the amount of data that needs to be copied over the network, resulting in much faster algorithms but it cannot be known in advance how many times combiners are called; it could be zero, one or multiple times to run.

In contrast with in-mapper combining, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers [5]. Therefore, in-mapper combining is more efficient than normal combiner and the proposed B+ tree based is embedded in this one for collection intermediate data which can mostly struggle in the network for further processes.

Either combiner or in-mapper combines keys/values pairs with the same key together. They may also some additional preprocessing of combined values. [7]. Although, using in-mapper could introduce the memory limitation, a counter variable can be set to solve it whenever it is time to spill the partial results of each map task. The process flow of combiner and in-mapper is illustrated in Figure 1.

4.2 Using Pair or Stripe

With the problem of building word co-occurrence matrices from large corpora, a common task in corpus linguistics and statistical natural language processing, more complex strategies are applied to speed up map reduce processing. Two popular techniques: pair and stripes are used alternatively. Using pair approach, the Map Reduce execution framework guarantees that all values associated with the same key are brought together in the reducer. As for stripe way, all associative arrays with the same key will be brought together in the reduce phase of processing [5].

The pair approach is easy to complex but generates an immense number of key-value pairs compared to the stripe approach which comes with more serialization and deserialization overhead with the former one. But, both algorithms can benefit from the use of combiners or in-mapper.

4.3 Local Aggregation with Merge Sort

As for Hadoop, the underlying mechanism used for handling intermediate data in a Map Reduce computation is essentially via a parallel merge-sort [9]. The partial output produced by each mapper is periodically sort and spills the data to a file on disk. Then, after all finish, those intermediate data are transferred to the reducer which merges the data (using a disk-based merge if necessary) to produce the final output. But, whenever the map task emits data, the amount of partial results grows larger and sometimes, that exceed RAM limit which is skew in the output size of the map task and can become scalability bottleneck.

5. B+ tree based Local Aggregation

Actually, as input data size increases, intermediate and output data sizes are also quite large and, finally, the result-combining step adds a large overhead to the total run time. Parallel data flow frameworks are very sensitive to system parameters that users are expected to tune. With Hadoop, a user has to choose the number of reduce tasks for a Map-Reduce job and tune the parameters related to sorting the output of a map task. To lower the disk overheads, it is crucial to minimize.

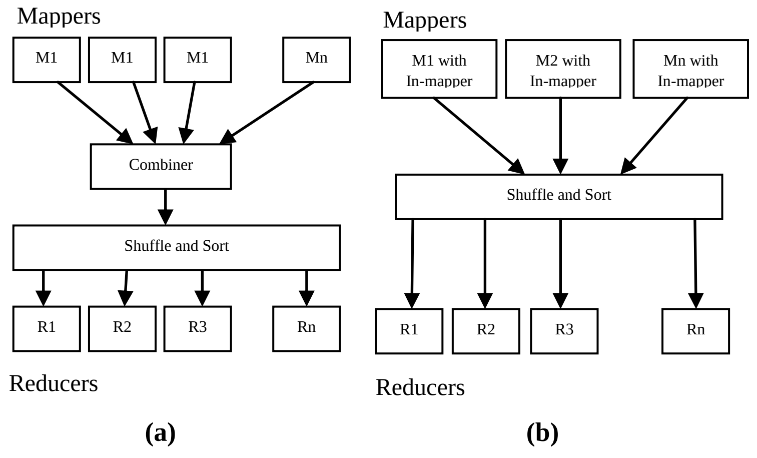

Figure 1: Map Reduce with (a) Combiner, (b) In-Mapper

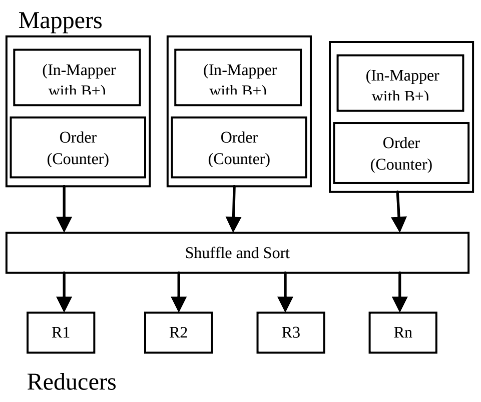

Figure 2: Map Reduce with Proposed B+ Tree based Local Aggreagtion

the number of disk seeks. Fewer disk seeks translates to increasing the amount of data read per seek.

In the proposed system, Hadoop is a base framework and replaced the algorithm usage of shuffle and sort state of map reduce phase with modified B+ tree. After each mapper produces key/value pair, the intermediate data is collected with modified B+ tree initially and then associated key pairs are grouped together according to their weights. When all the mappers finished, the collections are transferred to the reducer. The illustration of the proposed process flow is shown in Figure 2.

5.1 Weight-based Allocation

Frequent data insertion and deletion can make the system different from the current state and it also requires the system to be load balance. Since B+ tree is self-balanced structure, which is suitable to weight-based object allocation for the proposed system. Accessing the object (key) is only to use the *object id* which is calculated based on particular weights and locates where to place in the tree. All objects IDs (keys) in the proposed system are organized by a B+ tree. The object index is derived from the calculation of its weight. Therefore, a single attribute < *object id*> is supported as an index which points to the actual location of the object in the cluster. Since B+ tree consists of two types of nodes: internal nodes and leaf nodes. Internal points to other nodes in the tree whereas the leaf node points to actual data using data pointers. In addition, the leaf node also contains an additional pointer, called the sibling pointer, which is used to improve the efficiency of certain types of search [13]. The leaf node stores the actual address of the object and internal nodes points to the index of the child nodes which are allocated by their weight.

5.2 Proposed B+ Tree

As for data insertion in the existing media, the traditional B+ tree can take at least two steps (levels) for initial and then, the tree gets longer and longer as the number of stored item increased. No one can tell how large the tree depth and can result in delaying access time.

In afford to reduce the load traffic resulting from large data traversing; the proposed system makes little changes to the original tree to be getting better performance. Since keys are arranged by their weight, it must be needed to know which key (object) in the present system has the nearest value to the new key to place. Step 1 is taken new object into account for searching nearest neighborhood. The second step of the algorithm is not different from the original view. The bucket found in the previous step is checked whether it is full or not. If the condition is “ok” (not full), data is only placed. Otherwise, bucket separation is performed and new leaf’s smallest key is addressed into the parent node. After passing two steps, the next one is only considered for increasing order of the parent node

Table1: Proposed B+ Tree-based local aggregation algorithm

Insertion

Step 1: Perform a search to determine what bucket the new record should go into.

Step 2: If the bucket is not full, add the record.
Otherwise, split the bucket.
Allocate new leaf and move half the bucket's elements to the new bucket.
Insert the new leaf's smallest key and address into the parent.

Step 3: If the parent is full, check whether high is within range.
If high is ok, split it too.
Add the middle key to the parent node
Repeat until a parent is found that need not split.
If the root splits, create a new root which has one key and two pointers.
If high is not ok, promote the value of order. Go to Step 2.

which is a major contribution of the proposed system for achieving high data available and removing unnecessary network traffic for frequent data insertion and searching time in the shuffling of intermediate data. In those cases experienced in the former B+ Tree, parent nodes is also split again and creates new keys and cause the tree level high. This can be searching time further and further in parallel with the node number increased. The proposed B+ Tree as illustrated in Table 1 simplifies it by only raising the tree order according to power of 2-based form. Having increased the order by 2 power, much more parent nodes as well as child nodes can be handled and also the time complexity remains stable.

5.3 Resource Expansion with Proposed B+ Tree

Beginning from the base 2 of the order (b), there must be at most one search key value (b-1) and two child pointer (b<=n<=b) for each internal node as well as the root node. When new objects are requested to be stored, the tree requires expanding. However, node allocation starting from 2^1 is too short to be explained and thus, it will be more clear in the example with 2^2 of the order value. Therefore, it raised the order by power of 2-based form and now, the order value becomes 2^2 (b=4) and the number of children grows up to 4 whereas 3 for search key value in each internal node.

Table 2: Complexity comparison of mrge sort and proposed B+ tree algorithm

Input Data (Key/Value)	Merge Sort	Modified B+ Tree Sort
Input=2	1.0	1.0
Input=4	2.0	1.7924812503605783
Input=8	3.0	1.9357849740192017
Input=16	4.0	1.9767226489021297
Input=32	5.0	1.990839262077375
Input=64	6.0	1.9962133205833195
Input=128	7.0	1.9983835266817382
Input=256	8.0	1.9992941796073573
Input=512	9.0	1.9996866089930692
Input=1024	10.0	1.9998590429745329

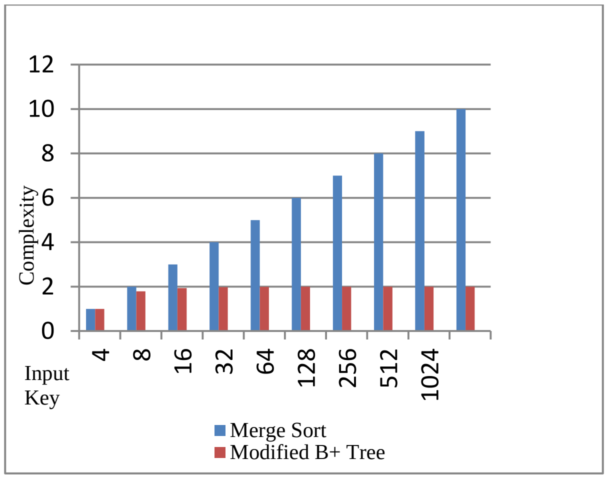

Figure 3 Complexity comparison of merge Sort and proposed B+ tree based data sorting

In ordinary B+ tree model, if b is threshold for each node to have, then adding new record which exceeds the specified threshold makes node splitting in two conditions, either of parents or child node. Instead of splitting parent node, increasing the order size can be the level of complexity constant and no high is raised.

5.4 Limiting Memory Usage

When using in-mapper combining approach, immense numbers of intermediate data are produced each map task and the associative array holding the partial term counts will no longer fit in memory [5]. One solution is to block input key-value pairs and flush in memory data structures periodically. Instead of emitting intermediate data after processing each key value pair, there should be set a counter variable (n) that keeps track certain number of key value pairs that have been processed and invoke to spill. Since the proposed B+ tree expands whenever the order size increases, the order variable can use as a threshold that determine what time intermediate data has to be spilled to disk. No one step is blocked due to not having run out of memory space.

6. Complexity Analysis

6.1 Merge-sort

In sorting n objects, merge sort has an average and worst-case performance of O (n log n). With terabytes, the time complexity for sorting those amounts of data can result in processing slow. As long as the amount of key value increase to sort, number of times to process (merge and sort) will be frequently and that may takes longer time and finally result in access speed slow within the network as shown in Table 2.

6.2 B+ Tree

B+ tree performance is logarithmic with respect to the number of height. The total time complexity of the tree takes $O(\log_b n)$ in general for b order of the tree with h level index. When increasing height, the depth becomes longer and it is taken time complexity more complicated. Contrary from traditional B+ tree, the proposed approach is emphasized on the depth of the tree which becomes skew in the original one. By controlling the order and height, the proposed approach can perform better than the existing one.

6.3 B+ Tree Vs Merge Sort

However, Map Reduce can be beneficial for large scale data processing; the algorithm applied plays a major role to make the model efficient. As shown in Table 2 and Figure 3, it is tested with variable input size and analyzed using two approaches: B+ tree and merge sort complexity theoretically. According to the statistical data analysis, whenever the input key value produced of each map task becomes larger, total time (local aggregation of intermediate data) complexity of conventional merge-sort of Map Reduce takes longer time gradually after one another than the proposed approach. Therefore, it is proved that simple merge sort cannot parallel to the modified B+ tree for efficient data sorting especially terabytes to petabytes of data.

7. Conclusion

Although Hadoop has benefited from the use of map reduce data processing, careful handling of intermediate data is needed in order to reduce network traffic which can impact access time heavily. Variety of local aggregation techniques: combiner, in-mapper etc. are introduced to overcome this issue. Coupling the modified B+ tree with existing in-mapper combining approach can prove to perform better than regular usage of merge sort.

References

[1]. Abhishek Verma, Nicolas Zea, Brian Cho, Indranil Gupta, Roy H. Campbell, Breaking the MapReduce Stage Barrier, 2010 [3] MapReduce in Practice
[2]. Brad Hedlund, Understanding Hadoop Clusters and the Network. http://bradhedlund.com
[3]. David Silberberg, MapReduce and The Cloud.
[4] Jeffrey Dean and Sanjay Ghemawat, A Map Reduce Flexible Data Processing Tool, January, 2010.
[5] Jimmy Lin and Chris Dyer, Data Intensive Text Processing with MapReduce, April 11, 2010.
[6]. Jorda` Polo, David Carrera, Yolanda Becerra, Jordi Torres and Eduard Ayguade, Performance Management of MapReduce Applications, September 2009.
[7]. Maria Jurcovicova, MapReduce Questions and Answers Part I. http://javacodegeeks.com/2012/05/mapreduce-questions-and-answers-part-1.html.
[8] Pietro Michiardi, Hadoop MapReduce in Practice, 2011.
[9] Sriram Rao, Raghu Ramakrishman, Mike Ovsiannikov, Damian Reeves. SALFISH: A FRAMEWORK FOR LARGE SCALE DATA PROCESSING, 2012.
[10] Tom White, Hadoop: The Definitive Guide, 2009.
[11]. Wang, C.C and Hsu, Y. Wofs: A Distributed Network File System Supporting Fast Data Insertion and Truncation, 2010.
[12]. Zhou, G., Yuan, L. and Chen, J. B+ Tree Management Method of Object Attributes for Object-Based Storage, 2007.
[13].http://www.mec.ac.in/resources/notes/notes/ds/bplus.htm
[14]. http://www.wikipedia.bplus.html