ENHANCED SHUFFLE GROUPING FOR STREAM PROCESSING IN BIG DATA ANALYTICS

Sathyapriya N , Sandhiya D, Saisree K, Moorthi K
Department of Computer Science and Engineering,
Jansons Institute of Technology.
sathyapriya.rajan@gmail.com, sandhiya.d96@gmail.com, saisreekrishnan@gmail.com,
moorthicse@gmail.com

Abstract

Stream processing systems perform analysis on continuous data streams. A stream processing application contains data operators and streams of tuples containing data to be analysed. Grouping function strategy routes the tuples towards the operator instances. Shuffle grouping is a technique used by stream processing frameworks to share input load among parallel instances of stateless operators. With shuffle grouping each tuple of a stream can be assigned to any available operator instance, independently from any previous assignment. A common approach to implement shuffle grouping is to adopt a round robin routing policy, a simple solution that adapts well as long as the tuple execution time is constant. In shuffle grouping each operating instance gets equal number of tuples. However, such assumption rarely holds where execution time strongly depends on tuple content. As a result, parallel stateless operators within stream processing applications may experience unpredictable unbalance that causes undesirable increase in tuple completion time. Proactive Online Shuffle Grouping (POSG), a novel approach to shuffle grouping aims at reducing the overall tuple completion time. POSG estimates the execution time of each tuple, enabling a proactive and online scheduling of input load to the target operator instances. Sketches are used to efficiently store the otherwise large amount of information required to schedule incoming load.

Keywords: real-time data, Stream Processing, Shuffle grouping, Count Min sketch Algorithm, Greedy Online Scheduler.

I. INTRODUCTION

Any data that exceeds our current capability of processing can be regarded as "big". Big Data are high-volume, high-velocity, and high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization. One can take data from any source and analyze it to find answers that enable cost reductions, time reductions, new product development and optimized offerings. Data and analytics centricity is a state of being where big data analytics are available to all the parts of the organizations that needed them. With the underlying infrastructure, data streams and tool sets are required to discover valuable insights solve actual business problems. Continuous real time data arrives at very high rate from various remote sources which should be processed and analysed simultaneously.

Analytics falls along a spectrum; one end is batch analytical applications (Hadoop based workloads) which are used for complex, long running analyses. They tend to have slow response time. Real-time analytical applications forms the other end of the spectrum. It provides faster light weight analytics with low latency and high availability requirements by offering distribution computation facilities. Several such applications include share market and stock trading, Web traffic processing, Network monitoring, Sensor based monitoring, Click streams, Social media and log data analyses etc., Issues in real time analysis include scalability, processing storage, continuous streaming data, need for special computational powers.

II. DISTRIBUTED STREAM PROCESSING SYSTEMS

Stream processing is the real-time processing of data continuously, concurrently, and in a record-by-record fashion. It treats data not as static tables or files, but as a continuous infinite stream of data integrated from both live and historical sources. It continuously performs mathematical or statistical analytics using "continuous queries". Stream processing takes the inbound data while it streams through the server. It also connects to external data sources, and allows applications to incorporate selected data into its flow, and can update an external database with processed information. Input arrives very rapidly and there is limited memory to store the input. Algorithms have to work with one or few passes over the data, space less than linear in the input size or time significantly less than the input size.

Distributed Stream Processing Systems(DSPS), offer a highly scalable and dynamically configurable platform for time-critical applications. DSPS have the ability to process huge volumes of data with very low latency on clusters of commodity hardware. Streaming applications are represented by Directed Acyclic Graphs (DAG), where vertices called processing elements, representing operators and edges called streams, representing the data flow from one processing element to next. DSPS consists of large number of Processing Nodes (PN). Applications are deployed on PNs as a network of operators or Processing Elements (PEs).The grouping in DSPS is usually implemented by partitioning the stream by several methods such as key grouping, shuffle grouping, field grouping, partial key grouping, global grouping, none grouping, direct grouping, local grouping and more.

Data streams are feed into Distributed Stream Processing Systems (DSPS). Each processing element perform some operation on the input stream such as filter, aggregate, correlate, classify and transform. In order to carry out the computation, the PE uses computational resources of the PN on which it resides. These resources are finite, and are divided among the PEs residing on the node. The output of this computation could alter the state of the PE, produce an output with the summarization of the relevant information derived from (possibly multiple) input streams and the current state of the PE.

Apache storm, a Distributed Stream Processing System is scalable, fault-tolerant and is easy to set up and operate. Storm's use cases include real time analytics, online machine learning, continuous computation, distributed RPC, ETL and more.It relies on a cluster of heterogeneous machines. According to Storm architecture, a machine, called a worker node is composed of workers. Each worker contains a variable number of slots. A slot is an available computation unit which has dedicated CPU re-sources. When an operator is affected to a slot, it is encapsulated in an executor. According to our generic architecture, processing units are equivalent to workers. Each operator corresponds to an executor.

Continuous queries are represented as user-defined workflows, denoted as topologies. These topologies are also workflows but vertices belongs to two main categories: Spout and Bolt. Spouts are data transmission nodes and can be conceptualized as multiplexers. They provide an interface between sources and processing environment. After getting connected to one or many sources, they transmit data streams to one or many Bolts. Each Bolt execute a user-defined operator which is considered as atomic. Storm does not include primitives but provides programming patterns. Basically, Storm does not support stateful operators, so naturally does not support window incremental processing, but they can be added through API extension. Allocation of executors on workers is achieved by a scheduler minimizing CPU and memory usage. Storm targets applications handling huge volume of data like social data management.

III. PROACTIVE ONLINE SHUFFLE GROUPING

Proactive Online Shuffle Grouping (POSG), an approach in shuffle grouping that aims at reducing the overall completion time of incoming tuples by accurately scheduling it on

available operator instances thereby avoid imbalances. POSG is the first solution that explicitly addresses the problem of imbalances in parallel operator instances under loads characterized by non-uniform tuple execution times.

The basic idea behind POSG algorithm is simple. By measuring the amount needed by each operating instances to process every tuples, we can schedule incoming tuples accordingly, in order to minimize the completion time. This approach works in the case of non-trivial streaming settings. However, POSG uses sketches in order to keep track of huge amount of information regarding the tuple execution time and then applies a greedy online multiprocessor scheduling algorithm to dynamically schedule incoming tuples to operator instances. Concurrently, the status of each instances is monitored in a smart way which in turn used to detect possible changes in the input load distribution that can be adapted coherently. Therefore, POSG improves performance in terms of tuple completion time.

To be clearer, if the execution time of each tuple on available operator instance is known, it is easy to schedule the execution of incoming tuples on such instances with the aim of minimizing average per tuple completion time at the operator instances. However, the common way of calculating the execution time of each tuples is by building cost model for tuple execution and then utilizing it to schedule the incoming load proactively. But building an accurate cost model requires large a-priori knowledge on the system. Furthermore, once the system is built, it is hard to handle changes in the system or input stream characteristics at runtime. Another alternative is to periodically collect the load of operator instances at scheduler, which becomes reactive scheduling, where the input tuples are scheduled on the basis of previous load state of the operator instances.

Therefore, POSG computes the estimation by summing the estimation of execution time of each tuples assigned to each operator instances. A greedy scheduling algorithm is then fed with estimations for all the available instances, which can be enabled by building a sketch at each operator instances. That sketch will track the execution time of the tuples it process. Any change in the stream or instance(s) affects the tuples execution time on some instances. Therefore, the concerned instance(s) will forward the updated sketch to the scheduler, which allows the scheduler to estimate the tuples execution time correctly. This solution does not requires any a-priori knowledge. This approach is also adaptable to continuous changes in the input distribution or on the instances load characteristics. Therefore, this solution is proactive and avoids imbalance rather than detecting and then attempting to correct it.

IV. COUNT-MIN SKETCH ALGORITHM

The Count-Min (CM) Sketch is a compact summary data structure capable of representing a high-dimensional vector and answering queries on this vector with strong accuracy guarantees. Such queries are at the core of many computations, so the structure can be used in order to answer a variety of other queries, such as frequent items (heavy hitters), quantile finding, join size estimation, and more. Since the data structure can easily process updates in the form of additions or subtractions to dimensions of the vector (which may correspond to insertions or deletions, or other transactions), it is capable of working over streams of updates, at high rates. The data structure maintains the linear projection of the vector with a number of other random vectors. These vectors are defined implicitly by simple hash functions. Increasing the range of the hash functions increases the accuracy of the summary, and increasing the number of hash functions decreases the probability of a bad estimate. These trade-offs are quantified precisely below. Because of this linearity, CM sketches can be scaled, added and subtracted, to produce summaries of the corresponding scaled and combined vectors.

The CM sketch is simply an array of counters of width w and depth d, CM[1, 1] . . . CM[d, w]. Each entry of the array is initially zero. Additionally, d hash functions

$$h1 \ldots hd: \{1 \ldots n\} \rightarrow \{1 \ldots w\}$$

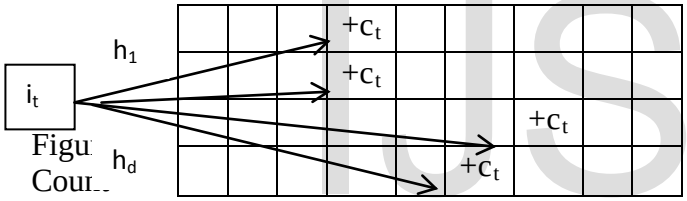
are chosen uniformly at random from a pairwise-independent family. Once w and d are chosen, the space required is fixed: the data structure is represented by w_d counters and d hash functions (which can each be represented in O(1) machine words.

Update Procedure: Consider a vector a, which is presented in an implicit, incremental fashion. This vector has dimension n, and its current state at time t is a(t) = [a1(t), . . . ai(t), . . . an(t)]. Initially, a(0) is the zero vector, 0, so ai(0) is 0 for all i. Updates to individual entries of the vector are presented as a stream of pairs. The ' t' update is (it, ct), meaning that

ait (t) = ait (t − 1) + ct

ai (t) = ai 0 (t − 1) i'= it

This procedure is illustrated in Figure 1. In the remainder of this article, t is dropped, and the current state of the vector is referred to as just a for convenience. It is assumed throughout that although values of ai increase and decrease with updates.



Figu
Cour
Min-Sketch

Each item i is mapped to one cell in each row of the array of counts: when an update of ct to item it arrives, ct is added to each of these cells each ai ≥ 0. The Count-Min sketch also applies to the case where ais can be less than zero, with small factor increases in space. Here, details of these extensions are omitted for simplicity of exposition (full details are in [5]). When an update (it, ct) arrives, ct is added to one count in each row of the Count-Min sketch; the counter is determined by hj. Formally, given (it, ct), the following modifications are performed: ∀1 ≤ j ≤ d : CM[j, hj (it)] ← CM[j, hj (it)] + ct Because computing each hash function takes O(1) (constant) time, the total time to perform an update is O(d), independent of w. Since d is typically small in practice, updates can be processed at high speed.

V. GREEDY ONLINE SCHEDULER ALGORITHM

A problem in load balancing is scheduling the important tasks to the identical machines and thereby minimizing the execution time i.e., the Multiprocessor Scheduling problem. However, the algorithm that is optimal for minimizing total flow time often starves some individual tasks. This problem becomes more complex when these tasks have priorities. To overcome these issues, in POSG, greedy online scheduler algorithm is used to schedule online independent tasks on non-uniform machines aiming to minimizing the average per task completion time. Online means the scheduler does not know the sequence of tasks it is going to handle in previous. The Greedy Online Scheduler algorithm assigns the incoming task to the instance having less load.

In POSG, each operator instances maintains two count-min sketch matrices, where the first matrix monitors the frequency of rules and the second matrix maintains the cumulative execution time of rules. Both the matrices shares same size and hash function. Both the matrices are updated each time after tuple execution by the operator instances.

VI . CONCLUSION

In this paper, various approaches for stream processing in stateless operator instances have been discussed. Shuffle grouping concepts are briefly discussed along with the current issues. Alternatives for the above stated problems such as cost model estimation and reactive scheduling were stated along with their drawbacks. Then, Proactive Online Shuffle Grouping, a novel approach to shuffle grouping aimed at reducing the overall tuple completion time by scheduling tuples on operator instances on the basis of their estimated execution time was introduced and discussed. POSG makes use of sketch data structures to keep track of tuple execution time on operator instances in a compact and scalable way. This information is then fed to a greedy scheduling algorithm to assign incoming load. Thus, POSG provides important speedups in tuple completion time

when the workload is characterized by stateless operator instances.

VII. REFERENCES

[1] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems",in Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, ICDCS, 2006.

[2] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms,55, 2005.

[3] Nicol_o Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, Bruno Sericola, " Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems", HAL archives, 2016.

[4] M. A. U. Nasir, G. D. F. Morales, D. G. Soriano, N. Kourtellis, and M. Serani. The power of both choices: Practical load balancing for distributed stream processing engines. In Proceedings of the 31st IEEE International Conference on Data Engineering, ICDE, 2015.

[5] M.G.Noll, "Implementing Real-Time Trending Topics With a Distributed Rolling Count Algorithm in Storm-2013", www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm.

[6] Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, Yves Caniou, "Parallel and Distributed Stream Processing: systems Classification and Specific Issues", HAL archives,2015.

[7] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS, 2015.

[8] Supun Kamburugamuve," Survey of Distributed Stream Processing For Large Stream Sources", Phd Thesis,2013.

[9] Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams.In: Proceedings of 20th International Conference on Data Engineering, 2004, pp. 350–361.IEEE, Boston (2004)

[10] Ryvkina, E., Maskey, A.S., Cherniack, M., Zdonik, S.: Revision processing in a stream processing engine: a high-level design. In Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE'06, pp. 141–141. IEEE, Washington (2006)

[11] The Apache Software Foundation. Apache Storm(http://storm.apache.org).