The design and implementation of a system to detect and filter large sessions automatically

0. Abstract

Large sessions waste a lot of computing resources and extend the delivery time of MapReduce jobs. Automatically detecting and filtering large sessions guarantee cleaner data and better system without extra human involvement. Our sytem maintains a large session guid blacklist and updates it periodically. The experimental results show that the job performance is on average 35% better than before.

1. Introduction

The following two concepts are the key abstractions in our tracking system:

- Event: the abstraction of one user interaction. For example, when a user views a item page will generate one "ViewItem" event, which contains the id of this user, the id of this item as well as other useful information like IP and user agent.
- Session: the abstraction of the key information to describe a group of continuous events which contains the same "guid".

We call the process to generate session based on events as "sessionization". The whole process can be split into two steps:

1. Group the events with their guids and sort them in each group.
2. Check each group successively and extract common information as well as do some analysis.

"MapReduce" is very suitable for this process. We read the raw event in the "Map" stage and the framework will sort the events and shuffle to reduce tasks based on their guid. Then the "Reduce" stage handles the extraction and analyzing affairs.

We define a large session as one in which the event count is larger than a threshold.

According to the mechanism of sessionization, the same guid will be grouped to one reduce task. If one session is large, i.e., one guid has too many events, this reduce task will be much slower than other threads. A job has to wait for this task to return. This July, one of our sessionization jobs took over 2 hours and it should usually finish within 25 mins. We did a lot of queries and finally found that one single guid had more than 18 million events!

Large sessions waste a lot of computing resources and extend the delivery time of our jobs.

So, how are the large sessions born? Large sessions usually have two sources. First, some abnormality in applications. If one application sets the guid in a wrong way or replays one guid due to bugs, different users may share one single wrong guid and the large session is generated. This is harmful as all these users' data gets contaminated. Second, some bot thread replays one guid again and again, for example, a crawler downloads the view item pages on our website with one guid. This kind of data is useless for our analysts and should be filtered.

Analyzing historical data, we find that tens of new large session guid's are detected every day. It is very tedious to update code or configuration of our jobs manually every time we finding a new large session guid. Therefore, we design and implement a system to automatically detect large session guid's and filter these useless events.

The basic idea of our system is to maintain a blacklist in a configuration file, and this file saves the detected large session guid's and can be loaded in the MapReduce job. MR job can then filter these events to protect our MR system. The MR job can update this file once it detects a new large session.

Designing such a system is not a trivial matter. We encounter two key challenges. First, more than 2000 mappers and 500 reduce tasks are running concurrently in our job and each of them may detect large sessions, so we must reduce the conflicts without increasing redundancies. Second, by analyzing the historical data we find that a large part of large session guid's don' t appear repeatedly, at the same time, we can always detect new large session guid's every hour. The size of blacklist increases over time; in our experiment, the increasing size is about 2KB every hour. So, we must control the size of the blacklist file.

For the first challenge, we delay the committing of detected large session guid into the " cleanup" method of the map task while always checking the existence of these guids before the committing. For the second challenge, we design and implement an LRU cache for the blacklist.

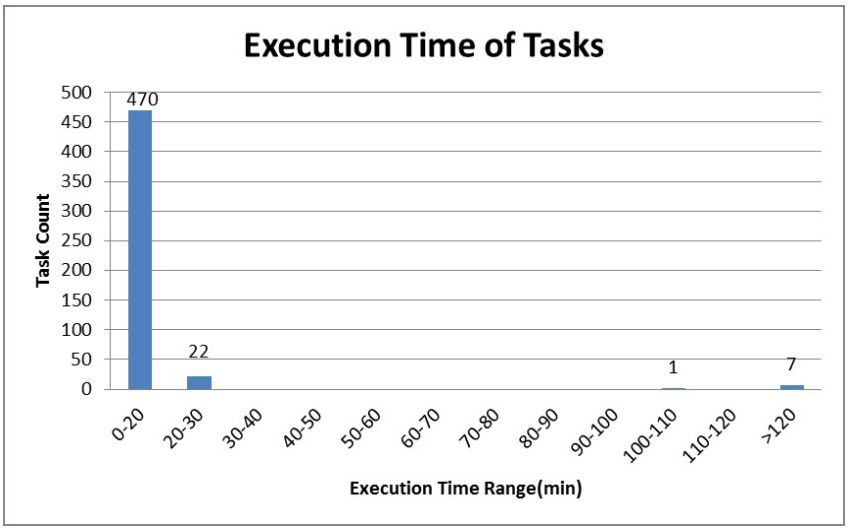
We have rolled out this system to production for several months with good results. The job execution time is reduced by 35%, and 99.4% of the conflicts when committing large session guid's are below 3. Thanks to the LRU mechanism, the blacklist is stable under 20KB.

The rest of this paper is organized as follows. Section 2 introduces the motivation of our system. In section 3, we introduce the architecture of our system as well as the functions of key components. Section 4 gives the implementation details of our system. We exhibit the experimentation results in Section 5. Section 6 concludes the paper.

2. Motivation

2.1 The harm of large sessions

The following picture shows the excution time of reduce tasks in a MR job which suffers the large session.



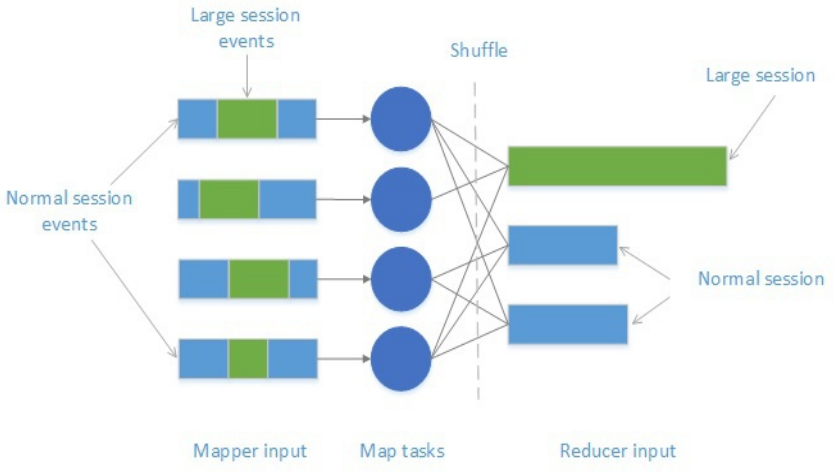
Suppose the job has 500 reduce tasks. 470 tasks finish within 20 mins. There are 8 large session guid's in the source files and they are shuffled to 8 different reduces tasks. One task takes more than 1 and a half hours and the left 7 tasks take more than 2 hours to handle too many events.

Shuffling too many events to one single reduce task has two main serious side effects. First, this task is much slower than other normal tasks. The finishing time of a MR job is restricted by the slowest task. So, the job deliver time has been extended. Second, maintaining too many data in memory wastes computing resources and may cause task fail and job fail.

2.2 The essense of large sessions

The essence of the large session is data skew.

As shown in the following picture, the events of some large session distribute evenly in the source files, but when they are shuffled according to the guids, these events will be grouped to one single task. The data skew is generated.

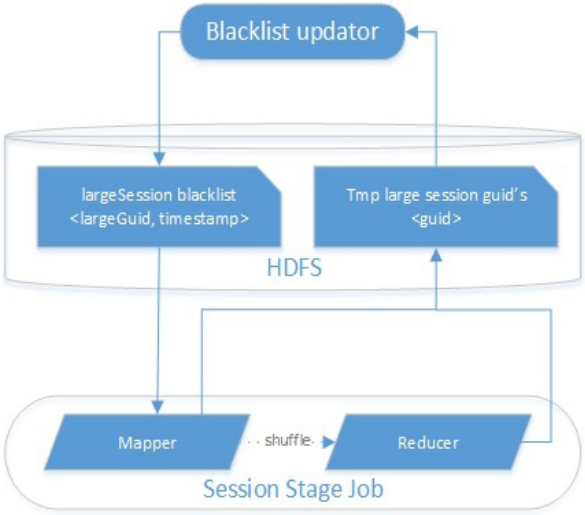


Grouping by guid of events is the basis of "sessionization" and we can not change it. This kind of data skew can not be avoided by redesigning the "partitioner". The only way is to filter them.

On the other hand, guid is related to timestamp and the large session guid's are changed dynamically. So, we can not use a static blacklist or always change the configuration file manually. We need a system to detect and filter the large sessions automatically.

3. Architecture

The architecture of our system is as follows:



The key components in our system are as follows:

- Large Session Blacklist: Saves the large session guid's and the corresponding timestamp. This file is updated by the "Blacklist Updator" periodically and loaded by the map tasks of the "SessionStage Job".
- Tmp Large Session Guid Files: Saves the guids detected in the last round of the "SessionStage Job". Both the map tasks and the reduce tasks can commit new large session guid's.

- Blacklist Updator: Collects the temp large session guid's and updates the timestamp in the large session blacklist.

The "Session Stage Job" is an periodic MR job which performs the sessionization tasks. We use HDFS as shared storage.

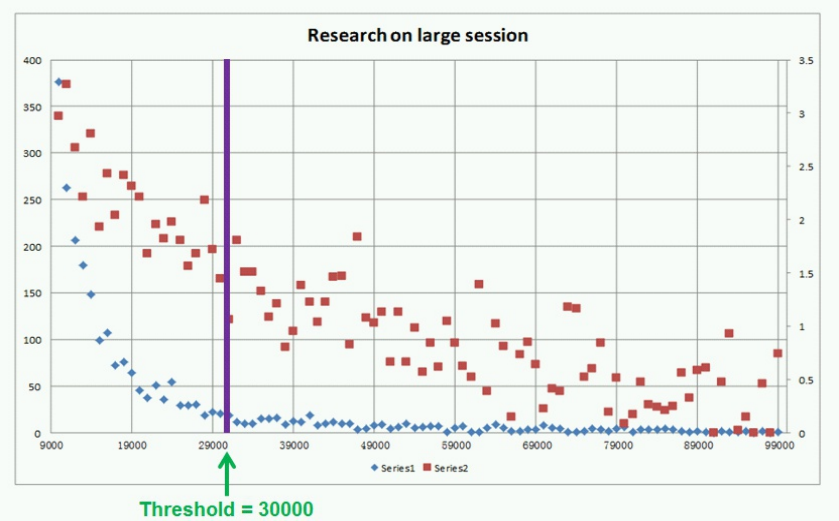
4. Implementation

This section introduces the implementation details of our system.

4.1 A research on the threshold of large session

There is a bot rule called "Too many events bot flag" which indicats that a session which has an event count larger than 10,000 is marked as "too many events bot". This threshold is different from the large session threshold. The bot sessions and events can be queried in our tracking system after sessionization and the data analyst may need these bot sessions to investigate the behaviour of bot. On the contrary, large sessions should be excluded from our system and these sessions and events are not to be stored and queried again. In other words, bot sessions and events are useful under particular conditions while large sessions and events should be filtered directly. We need two different thresholds.

The threshold of a large session should exclude the harmful events while being sure that these events are useless. In UBI, the threshold of event count in a large session is set to 100, 000. We did a research to find how they got this value and determine if we shall reduce this threshold to get a better performance. The following picture shows the plotted data.



- X-axis: The value of event count in Pulsar sessions, which means a group of sessions whose event count is under the same range. From 10,000 to 99,000. The points are multiple of 1, 000.

- Left Y-axis (series 1): The count of sessions whose event count is in the range of the point on X-axis. For example, the left-up point, series1 (10,000, 376) means that there are 376 sessions whose event count is between 9,000 and 10,000.
- Right Y-axis (series 2): The value of the average interval between events. This value is computed by ("average session duration" / "average event count"). For example, for the group whose event count is between 9,000 and 10,000, the average session duration is 29,760 seconds, the average interval between events of this group is 29,760/10,000 = 2.976.

As shown in this picture, the session count drops dramatically when the event counts range increases. When the event count is greater than 30,000, the count of this kind of sessions is only a united digit. At the same time, the average interval between events is under 1 second. The detail statistical information is as follows:

Event Count Threshold	Session Count	Average Event Interval (s)
<=10000	2263681857	2.98~4.39
10000~30000	1587	1.45~2.67
30000~50000	220	0.80~1.49
50000~100000	161	<1.14

As we can see, a human being can not generate this kind of session and the total amount of this kind of sessions is too small to do further common research.

So, we set the threshold to 30,000 events in our implementation.

4.2 LRU large session table

Guid is globally unique, and the guid of a large session won't be shared with a normal session later. In theory, we don't need to remove the large session guid's from the blacklist.

One practical problem is that new large session guid's are detected every hour and only part of them appear continuously. That means our large session blacklist becomes larger and larger while only part of them are useful.

Therefore, we add one timestamp field in large session blacklist. Now the blacklist is a map. The key is "guid" and the value is "timestamp". When some large session guid appears again, the timestamp will be updated. This "timestamp" field is like the "Update Time" field in a

database. Every hour, we check whether one guid is time out and remove these overdue ones from the blacklist to save memory. In our implementation, we set the timeout threshold to 72 hours (3 days) which means that if one guid hasn't appeared in the last 72 hours, it will be removed from the blacklist until it is proved to be a large session guid again.

The mechanism is just like an LRU cache.

4.3 Detect new large session guid's

Both mappers and reduce tasks can update the large session blacklist, but the semantic is different. The map tasks can only update the timestamp of existing large session guid's while the reduce tasks only submit newly detected large session guid's. The details are as follows.

4.3.1 Map task

The map tasks read the raw events and they can't get the total count of each guid, so they can't be used to detect new large session guid's. If one existing large session guid is read in a map task, we know that this large session guid appears again and we do not need to shuffle these events to reduce tasks. We just update the timestamp of this large session guid.

The raw events we read from source files are distributed uniformly and every map task has a chance to update the timestamp of large session guid's. We usually have more than two thousand map tasks in one job, so how to be sure that these tasks update the list orderly?

"Zookeeper" is the tipical solution for maintaining configuration information. However, adding the dependency of "Zookeeper" in a MR system brings too much efforts. We need to find easier options.

One solution is for every map task to write a specific folder or a specific file, for example, each map task creates a file named with the machine name and PID and writes the large session guid's it detected into this file, and then the "Blacklist updator" can load these files and do deduplication to update the final blacklist. Note that we don't care which map task the large session guid comes from, so, the drawback of this solution is these files which indicate the same guid are regarded as duplications. Suppose we have 2000 mappers and there are 100 large session guid's in raw events. We suppose each map task writes 80 large session guid's on average, then the "Lkp Job" will read 2000*80 guids while only 100 of them are useful.

The other solution is lazy committing and file locking. We create a hash set in the "setup" method of map task and if one guid is hit in the large session blacklist, this guid will be added into this set. Each map task commits the guids in this set to temp folder by "touching" an empty file named with the large session guid in the "cleanup" method. The task should check

whether the target file already exists in the temp folder. However, this solution can not guarantee no coflicts, shown in the following picture:

The green line and the yellow line represent the procedure of creating files by task1 and task2 respectively. At the start point of task1, task1 checks the file and finds that it doesn't exist, so it tries to create this file. At time point "a", the task2 checks the file and finds that it doesn't exist for the task 1 hasn't finished its creating procedure. So, task2 also tries to create the same file. At time point "b", task1 finishes creating the file and returns, however, task2 doesn't know this. At the end of task2, the HDFS throws an exception of "file already exists". The Hadoop framework makes sure that one file could only be created once under the same folder.

The lazy committing, which means uploading the large session guid's in the "clean up" method, could partly solve the conflicts, but not always. The conflict between files is handled in one try-catch block and we creat one counter in our job to log the count of confliction. We test with production data and find that there are only units digit conflicts each hour. We believe this doesn't slow down our job.

4.3.2 Reduce task

There are two possibilities when one session finishes computing in a reduce task. The first one is that the session ends in this hour (named as "sessionized session" in our system), and the second one is that the session doesn't end in this hour (named as "staged session" in our system) and the events of "staged session" should be copied to the next hour. In pulsar session schema, there is one field called "absEventCnt" which contains the total event count in this session. By comparing this count with large session threshold, we can detect new large session guid's.

As long as the "absEventCnt" reaches the threshold, no matter the session is "sessionized session" or "stage session", its guid should be added into the blacklist and the events in this session should be dropped. This saves a lot of effort because we don't need to copy the events of "staged session" hour by hour and filter useless events as early as we can.

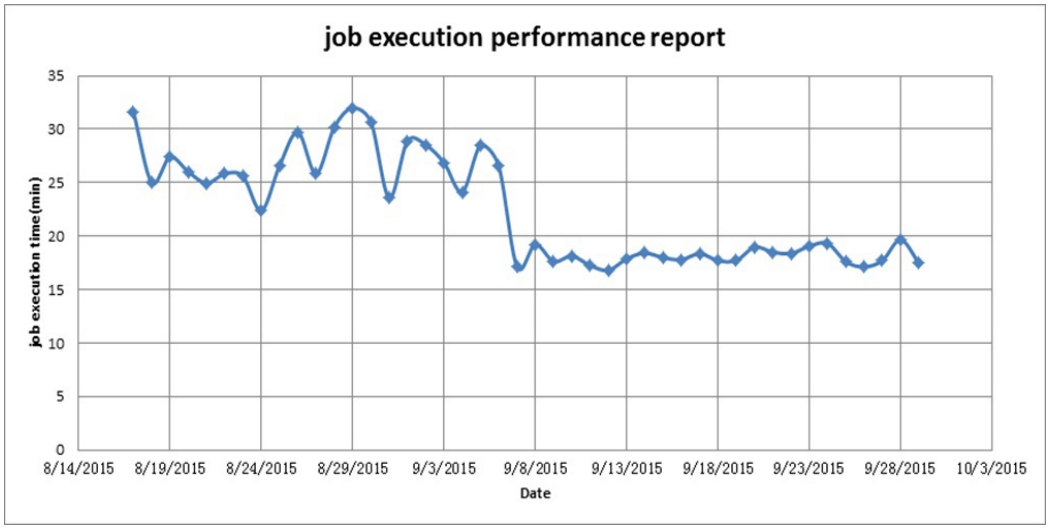
The reduce task uploads the newly detected large session guid at the end of the "reduce" method and it would not conflict with other reduce tasks for the reason that the events with the same guid are shuffled to the same reduce task.

5. Experiment

This section shows the experimental results of our system.
We rolled out this new feature to the production in September, 2015. Two points were considered when we designed the experiments and all the results come from real production jobs.

5.1 Job performance comparison

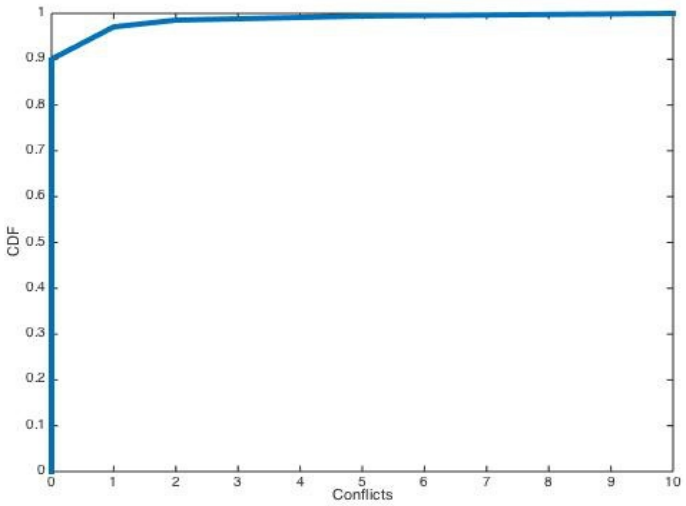
The key motivation for this system is to save job execution time. We queried the history of jobs using the restful API provided by Hadoop Eagle and get the average job duration every day for about one month. The following picture shows the job performance comparison.



Before adding this new feature, the average job execution time was 27.18 mins and the values had big fluctuations. After enabling large session detection and filtering, the average job duration dropped to 17.67 mins and the values are much stable. The job is 35% faster than before, and the excution time is more stable as well.

5.2 Conflicts when map tasks commit large session guid's

We designed and implemented a light-weight solution to handle the conflicts when map tasks commit large session guid's concurrently. To verify the performace of this solution, we added a counter in our map task and as long as one conflict is detected, the count will be increased by one. We collected the logs for 20 days and the "CDF" graph is as follows:



About 90% of jobs detect no conflicts and the conflicts of 99.4% jobs are under 3. Only 0.5 % of jobs have a more than or equal to 10 conflicts. This solution is much easier than using "Zoopkeeper" and the performance is acceptable as well.

6. Conclusion

This paper introduces the design and implemention of a system to automatically detect and filter large sessions. After enabling this new feature on production, the job performance increased by 35%. Both the robustness and the stability of our system have improved.