2015 IEEE International Conference on Cloud Engineering

SmartCache: An Optimized MapReduce Implementation of Frequent Itemset Mining

Dachuan Huang[1], Yang Song[2], Ramani Routray[2], Feng Qin[1]
[1]The Ohio State University [2]IBM Research - Almaden
{*huangda, qin*}*@cse.ohio-state.edu* {*yangsong, routrayr*}*@us.ibm.com*

***Abstract*—Frequent Itemset Mining (FIM) is a classic data mining topic with many real world applications such as market basket analysis. Many algorithms including *Apriori*, *FP-Growth*, and *Eclat* were proposed in the FIM field. As the dataset size grows, researchers have proposed MapReduce version of FIM algorithms to meet the big data challenge. This paper proposes new improvements to the MapReduce implementation of FIM algorithm by introducing a cache layer and a selective online analyzer. We have evaluated the effectiveness and efficiency of SmartCache via extensive experiments on four public datasets. SmartCache can reduce on average 45.4%, and up to 97.0% of the total execution time compared with the state-of-the-art solution.**

I. Introduction

Frequent Itemset Mining (FIM) [1] is a classic data mining topic whose goal is to extract out itemsets that appear above a certain threshold from data input. FIM is famous for its important role in market basket analysis. It provides foundation to association rule learning, because association rule's properties such as *confidence*, *lift*, and *conviction* are defined on top of frequent itemset's support value. Many algorithms such as *Apriori* [2], *FP-Growth* [3], and *Eclat* [4] were proposed to identify frequent itemsets in the last two decades. These algorithms take a *table* as their input. In the market business context, the table can be a market transaction log. Each row represents one customer transaction in the market, and each column represents one item the customer had purchased. The output is a collection of itemsets that represent popular combinations of items.

The market transaction scale, however, increases dramatically in recent years. For example, Amazon sold 306 items per second at maximum on Cyber Monday in year 2012, and sold 27 millions items in total on that day [5]; Taobao is an online transaction website in China, and its sales reached 5.7 billions of US dollars on November 11th, 2013 [6]. One single node cannot hold the data input in such scale altogether in the memory. What aggravates memory shortage problem is that, in the worst case, FIM algorithm can generate 2^n number of itemsets where n is the number of distinct items. These problems entail a distributed, data parallel, and scalable solution to FIM algorithms.

To address the above challenges, researchers have tried to adapt traditional FIM algorithms to the widely used cloud computing framework MapReduce [7] and its open source implementation Hadoop [8]. This is because MapReduce framework could leverage the computation power and storage capacity of many distributed commodity machines. More specifically, in MapReduce framework, a Map task processes one portion of data, called *split*, and passes the intermediate results to one or more Reduce tasks, which generate the final output. Some existing MapReduce implementations of FIM algorithm [9], [10] are direct translation from Apriori algorithm – they need the same number of MapReduce *phases*[1] as the number of loops in Apriori algorithm. In particular, the first MapReduce phase generates frequent itemsets with length one; the second MapReduce phase calculates the candidate itemsets from first phase's output, and then generates frequent itemsets with length two. The above process repeats until all frequent itemsets are found. This is a lengthy and expensive translation of Apriori algorithm. A two-phase MapReduce algorithm *MRApriori* [11] for FIM is proposed afterwards: infrequent itemsets in a split are filtered out in the corresponding Map task in the first phase, then all local frequent itemsets are union'ed together as global candidate itemsets, finally the second phase continues to filter out infrequent itemsets from candidates.

MRApriori algorithm, however, suffers from expensive scanning overhead problem in the second phase. If an itemset is frequent in one split after local FIM algorithm's counting, it will be treated as a candidate, but it has to be re-counted in every split in phase 2, including the split that has counted it before. The reason of this counting information loss is that the information is challenging to be used. A frequent itemset in one split does not imply that it is frequent in other splits. Since only frequent itemsets are reported, the sum of all reported counting values does not reflect the total occurrences of this itemset. Therefore, MRApriori chose to ignore all counting information from phase 1.

In this paper, we propose a new solution, called *SmartCache*, which continues to use the two-phase structure, but could avoid unnecessary scanning overhead in the second phase. SmartCache's effectiveness comes from our key observation: *if one itemset is very close to, although not above, the threshold, it might exceed the threshold in other splits*. This indicates that it can be beneficial to store more itemsets to avoid repetitive counting in the second phase.

In SmartCache, we first introduce a cache layer. This layer saves the counting information generated from phase 1 to HDFS [12] files, which are used by phase 2 to speed up its execution. Second, we design an online analyzer module to decide the cost-effective number of itemsets' counting information to be saved. The reason we need this module is because *phase 1's minimum support* (different from the user-provided global minimum support), which roughly reflects the amount of counting information to save, has a substantial influence to the total execution time. A low overhead, online

[1]In this paper, phase means a MapReduce job, not Map or Reduce phase.

978-1-4799-8218-9/15 $31.00 © 2015 IEEE
DOI 10.1109/IC2E.2015.12

module is needed to find an optimal value for this variable.

We evaluate SmartCache with extensive experiments on four publicly accessible and popular datasets *pumsb*, *accidents*, *T40I10D100K*, and *kosarak* [13]. The experiment results show that SmartCache can reduce on average 45.4%, and up to 97.0% end-to-end execution time compared with the state-of-the-art solution.

In summary, our contributions are threefold:

- We proposed a new MapReduce solution, called SmartCache, for the classic Frequent Itemset Mining problem. SmartCache is effective and has significant performance improvement over existing solutions;
- We designed a regression based online analyzer to find the optimal value for phase 1's minimum support, which is a variable that decides how many itemsets to save and further decides the I/O overhead;
- We conducted extensive experiments on four public and representative datasets, and compared SmartCache with two most recent works about the end-to-end execution time.

This paper is organized as follows. We first give the background information in Section II. Section III discusses a motivating example. Section IV discusses SmartCache in details. Experiment results are given in Section V. We have a discussion section in Section VI, and conclude our paper in Section VII.

II. Background: Traditional and Distributed FIM Algorithms

Frequent Itemset Mining algorithm's input is a dataset that consists of many *transactions*, or *rows*. Each row includes many *items*. *Itemset* is a set of items. *Absolute support* of one itemset is its number of occurrences. *Support* of one itemset is absolute support of this itemset divided by the number of rows. Once the *minimum support* is given, a *frequent itemset* is defined as an itemset whose support is bigger than or equal to it. For example, if the dataset contains two rows: 1) $Diaper, Toast, Beer$; 2) $Diaper, Beer$, and if the minimum support is 60%, then the frequent itemsets will be $\{Diaper\}$, $\{Beer\}$, and $\{Diaper, Beer\}$, whose absolute support values are all 2, and support values are all 100%.

There are three classic FIM algorithms that run in single node. Apriori [2] algorithm's main logic is a loop, where loop i generates frequent itemsets with length i. Loop $i+1$ calculates the candidate itemsets from output of loop i by using the following property: any subset in one frequent itemset must also be frequent. FP-Growth [3] algorithm establishes an FP-Tree by two passes of the dataset. Frequent itemsets are extracted from this data structure. Eclat [4] algorithm transposes the dataset into a new table where each row means one item's sorted transaction ID list. Frequent itemsets are found by intersecting two transaction lists.

Parallel FP-Growth (PFP) algorithm [14], which is implemented in Apache Mahout [15] scalable machine learning library, applies FP-Growth algorithm into Hadoop framework. However, as mentioned in paper BPFP [16], it suffers from load unbalance problem. In [17], the authors propose BigFIM method. While it is shown to outperform PFP and BPFP, BigFIM uses more than two phases of MapReduce jobs to compute. As will be shown in Section V, SmartCache has significant improvement over this solution.

Othman et al. [11] describes two ways of translating Apriori algorithm into MapReduce jobs. The first way is exhausting all possible itemsets in Map task, and letting the Reduce task filter out itemsets that are under minimum support. This approach is used by [18]. It suffers from an exploded intermediate results between Map and Reduce tasks. The second way is a direct translation from Apriori algorithm. It makes every loop inside Apriori algorithm a MapReduce job. This approach is used by [9], [10]. It also has a large amount of data to shuffle between Map and Reduce tasks [11]. Additionally, it has to pay for the extra MapReduce framework scheduling overhead. To address these issues, they proposed a better algorithm MRApriori, which uses two-phase structure.

There are two methods to further optimize MRApriori. First, we can prune the global candidate itemsets from phase 1 by some arithmetic properties of distributed FIM, then we can save the scanning overhead in phase 2. This approach is used by [19]–[21]. However, this approach assumes that each split's row size is known to each Reduce task in phase 1. This assumption can be achieved by either adding an extra MapReduce job ahead of algorithm, or piggybacking extra code in phase 1's Map task. The former causes an unnecessary MapReduce job overhead. The latter is not trivial to be extended to multiple Reduce tasks because a Map task has to broadcast its split size to every Reduce task. This broadcast task can be done by customizing hash partition function, but there is no universal way to get the function since a number of Map tasks and split size varies across different executions. Our method does not require this assumption. Second, we can re-use the itemset's frequency calculated in phase 1, such that phase 2 does not need to compute it again. This idea was partially explored by [21], whose solution is called *Zahra et al. solution* in our paper. However, their mechanism is static. Many itemsets' frequency values that could be re-used are not saved. Much space is left for improvement, as we will explain in Section III and compare it with SmartCache in Section V.

III. A Motivating Example

In this section, we will first briefly describe how MRApriori algorithm works, and describe what improvement Zahra et al. solution did to MRApriori algorithm, then explain why there is improvement space to Zahra et al. solution, and finally motivate our solution SmartCache.

Algorithm 1 MRApriori - Phase 1

```
1: procedure MAP(split, min_sup)
2:     L ← APRIORI(split, min_sup)
3:     for all ⟨itemset, abs_sup⟩ ∈ L do
4:         EMITINTERMEDIATE(itemset, abs_sup)
5:     end for
6: end procedure

7: procedure REDUCE(itemset, list abs_sups)
8:     EMIT(itemset, 1)
9: end procedure
```

Algorithm 1 and algorithm 2 show the two phases

Algorithm 2 MRApriori - Phase 2

```
1: procedure MAP(split, candidates)
2:     for all itemset ∈ candidates do
3:         abs_sup ← SCANCOUNT(itemset, split)
4:         EMITINTERMEDIATE(itemset, abs_sup)
5:     end for
6: end procedure

7: procedure REDUCE(itemset, list abs_sups, min_sup,
   total_rows)
8:     sum ← 0
9:     for all val ∈ abs_sups do
10:        sum ← sum + val
11:    end for
12:    if sum ≥ min_sup * total_rows then
13:        EMIT(itemset, sum)
14:    end if
15: end procedure
```

in MRApriori algorithm. These two phases are executed in sequential order. min_sup represents minimum support. abs_sup represents absolute support. $list\ abs\_sups$ means that variable abs_sups contains a group of abs_sup from Map tasks. $candidates$ is the output from phase 1. $total\_rows$, which is calculated in phase 1[2], means the total number of rows for the entire dataset.

Phase 1's Map task accepts the whole split as input, and runs the local FIM algorithm on it. This algorithm could be Apriori or FP-Growth. For correctness they make no difference because what we need is that the local algorithm can return local frequent itemsets. The Map task outputs each local frequent itemset as key, and its local occurrence as value. Phase 1's Reduce task directly outputs each itemset it got from Map task. The reason that Reduce task outputs *one* as itemset's value (line 8 in alg. 1) is because we cannot tell whether the sum of values is smaller than or equal to this itemset's real global occurrences at this stage. For example, if one itemset A is frequent in split 1 and its occurrence is 101, and itemset A is not frequent in split 2 and its occurrence is 99, then 99 will be dropped within split 2 locally, the sum of all reported values will be 101, which has no indication of itemset A's real occurrence. After this Reduce task, we get an output that has global candidate itemsets, which form a superset of the final frequent itemsets. The reason is that if the itemset is global frequent, it must be local frequent for at least one split. This mathematical property, which MRApriori's correctness relies on, can be proved by contradiction. Global candidate itemset are saved into DistributedCache in HDFS, because all of the phase 2's Map tasks need this as input.

Phase 2's Map task takes the same split and the global candidate itemsets as input. For each itemset, it scans the split row by row to count the occurrences, and outputs the itemset as key, the occurrences as value. Phase 2's Reduce task sums up all the values for each itemset, and does the final filtering. Itemset can only stay if its support is bigger than or equal to the given minimum support.

One inefficient part of MRApriori algorithm is that the counting information in phase 1's Map task is wasted. Itemset's occurrence has to be counted again in phase 2's Map task, and this is an expensive scanning operation. For example, if there are five Map tasks in phase 1, and three of them reported one itemset as frequent, then in phase 2, these three Map tasks will still need to count this itemset's occurrence again. Saving these intermediate itemsets' counting information so that phase 2's Map task does not need to count again becomes the guideline for both Zahra et al. solution and our solution SmartCache.

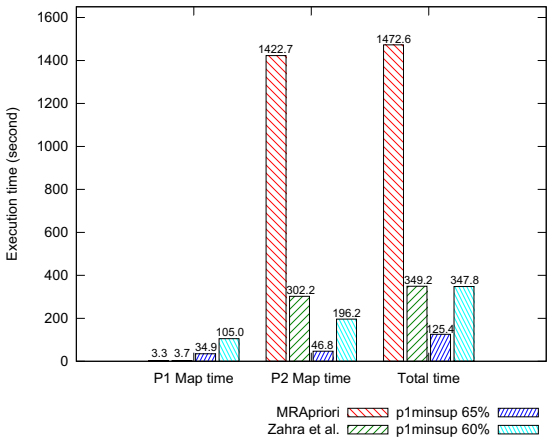


Fig. 1: Execution time comparison among MRApriori, Zahra et al. solution, and solutions that use lower phase 1's minimum support values. The dataset is *pumsb* and minimum support is 85%. Zahra et al. solution saves all itemsets that pass 85% threshold; the third bar saves all itemsets that pass 65% threshold; and the fourth bar saves all itemsets that pass 60% threshold. Every Reduce task takes less than 1.3 seconds, so they are omitted in this figure. $p1minsup$ represents phase 1's minimum support.

Zahra et al. solution proposed to re-use these counting information. Figure 1 shows phase 1's Map task execution time, phase 2's Map task execution time, and total execution time when the input dataset is *pumsb* and minimum support is 85% (we use this real workload $\langle pumsb, 85\% \rangle$ for all the figures in Section III and Section IV). From Figure 1, we can see that saving intermediate itemsets' counting information from phase 1's Map task can reduce the end-to-end execution time remarkably, about 76.3% compared with MRApriori (Figure 1 also tells us that this execution time reduction indeed comes from phase 2's Map task's execution time reduction), but we can also see that there is space for further improvement.

The important difference in our solution from Zahra et al. solution is that the latter is static while ours is dynamic. Zahra et al. solution, the same with MRApriori, uses minimum support as one of the input parameters in phase 1. However, we treat phase 1's minimum support as a *variable*, which can be made smaller than minimum support. The reason lies in our key observation: *even though one itemset does not exceed the minimum support threshold locally, if it is close enough to this threshold, likely it will exceed other splits' threshold. Therefore, saving its counting information is worth trying*. Since we do not change the global candidate itemsets, it is still a superset of the final frequent itemsets, the correctness of algorithm does not get affected. We can see from Figure 1 that

[2]Note $total\_rows$ variable is computed in phase 1 and used by phase 2, so this is different from letting every Reduce task in phase 1 know each split size. Its pseudo code is omitted here for brevity.

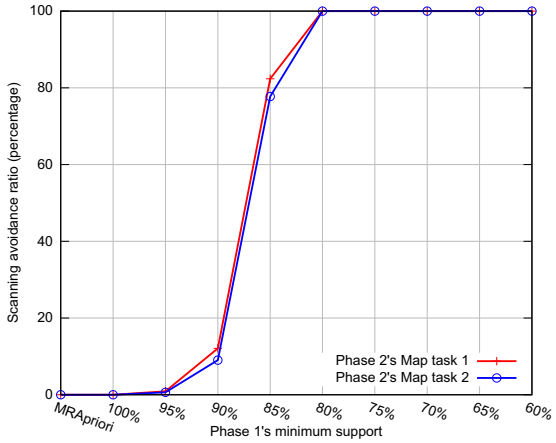


Fig. 2: Scanning avoidance ratio as a function of phase 1's minimum support. Inputs are $\langle pumsb, 85\% \rangle$.

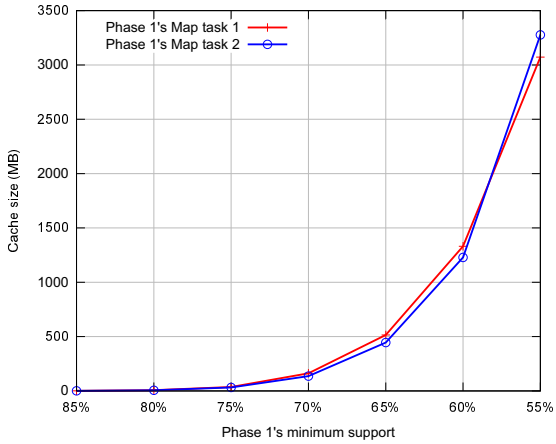


Fig. 3: Cache size as a function of phase 1's minimum support. Inputs are $\langle pumsb, 85\% \rangle$.

if we lower phase 1's minimum support to 65%, more itemsets' counting information will be saved, then it could bring even more benefits, about 91.5% time reduction compared with MRApriori, and 64.1% time reduction compared with Zahra et al. solution.

However, lowering phase 1's minimum support without restraint is not favorable. We define the intermediate itemsets' counting information saved by phase 1's Map task as *cache*[3], and define *scanning avoidance ratio* as the percentage of itemsets in global candidate itemsets that can be found in cache. This ratio can be calculated in phase 2's Map task. Figure 2 shows that the scanning avoidance ratio does not go higher after phase 1's minimum support is below 80%. Figure 3 shows that the cache size goes up non-linearly, which is because that FIM algorithm inherently generates 2^n itemsets (n is the number of distinct items) in the worse case. For example, if $\{a, b, c\}$ is accepted as a frequent itemset, then $\{a\}$,$\{b\}$,$\{c\}$,$\{a, b\}$,$\{a, c\}$,$\{b, c\}$ are also frequent itemsets. A huge cache size incurs high I/O overhead because we need

[3]*cache* means a HDFS file in this paper.

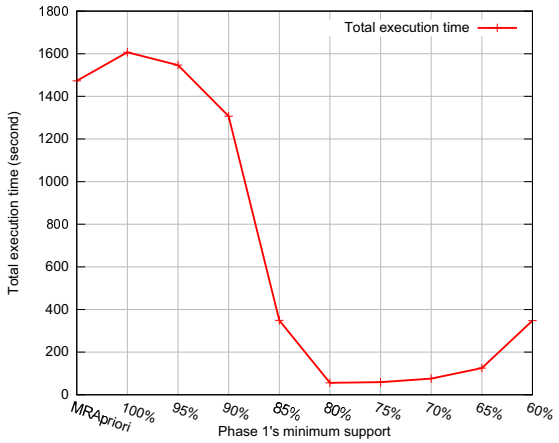


Fig. 4: Execution time as a function of phase 1's minimum support. Inputs are $\langle pumsb, 85\% \rangle$.

to write it into HDFS in phase 1 and read it from HDFS in phase 2. Figure 1 shows that if we lower phase 1's minimum support from 65% to 60%, phase 1's Map task time increases by 201% because of extra computation and writing overhead. Phase 2's Map task time increases by 319% because of extra reading overhead. The total execution time increases by 177%. The above analysis tells us that lowering phase 1's minimum support without restraint may even be harmful to performance.

To study how the execution time would change with phase 1's minimum support, we have tuned this variable to be different values. Figure 4 shows our result. For completeness purpose, we also tuned phase 1's minimum support to be above minimum support. From Figure 4, we can see as phase 1's minimum support goes lower, the performance first becomes better, because of higher scanning avoidance ratio; and then becomes worse, because of high I/O overhead. A large performance gap, which starts from above 1,400 seconds to below 100 seconds, shows up. In practice, there is no simple principle for users to guess the optimal variable value directly, and users usually do not have enough time to manually try every possible minimum support. Therefore, an online, low overhead solution that can find a variable value close to optimal one is desired.

IV. Our Approach: SmartCache

Figure 5 shows the workflow and software architecture for SmartCache. Besides its original output, which contains global candidate itemsets $G1$, phase 1's Map task also goes through a SmartCache data path. This new path has three components: local FIM algorithm interceptor, online analyzer, and cache generator.

Local FIM algorithm interceptor lowers the FIM algorithm's minimum support parameter, and uses the memory to hold all the returned itemsets; online analyzer analyzes these itemsets in real time, and identifies which itemsets to save; cache generator writes the identified itemsets into HDFS file, which is read by Map task in phase 2 as a *readonly cache*, for example, $C1$ or $C2$ in Figure 5. Whenever we want to count the occurrences of one global candidate itemset, we first look it up in the cache. If it is there, we save one round of scanning

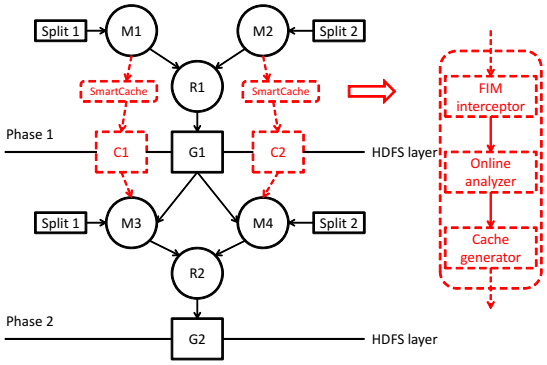


Fig. 5: Workflow and software architecture of SmartCache. Dashed (red) part is our change to MRApriori.

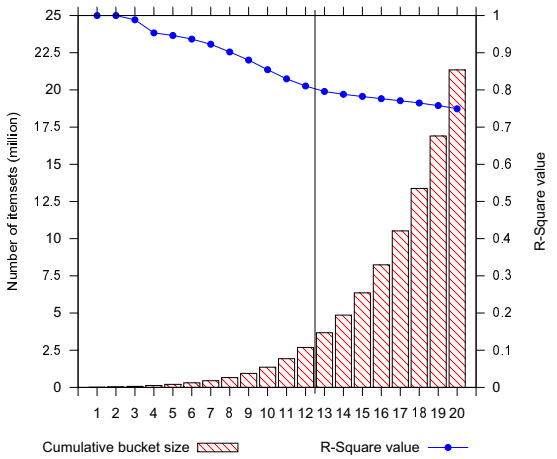


Fig. 6: SmartCache online analyzer's working logic in detail. It shows the cumulative bucket size trend, and the R-Square value trend. The statistics are from one Map task in phase 1. Inputs are $\langle pumsb, 85\% \rangle$.

overhead. After introducing our solution's architecture, the core problem is how to design an effective online analyzer, or concretely, how to decide how many itemsets should be saved into cache. As we have mentioned in Section III, it is not intuitive to make this decision.

It is ideal if we can establish a model about the cost and benefit, specifically, the cost of writing extra data into HDFS in phase 1 and reading these data from HDFS in phase 2, the benefit of saving the second phase's execution time. However, since the size of intermediate itemsets is not linear with phase 1's minimum support, as shown in Figure 3, it is hard to predict the cost function. Since the first phase's Map tasks are independent with each other, it is not easy to predict how many saved itemsets will be used in the second phase. Hence, it is hard to predict the scanning avoidance ratio in phase 2. Because of the above reasons, setting up a precise or even approximate cost and benefit model is challenging, if not impractical.

Our solution SmartCache can decide how many itemsets to save into cache *online*. We achieve this goal by detecting when cache size starts to grow non-linearly, and only saving itemsets before the turning point. First, we lower the minimum support in phase 1 by *slack* (between 0 and 1), then there is a *slack range*, which starts from *minimum support* and ends at *minimum support* * *slack*. The slack range is divided into *buckets* number of buckets. We change the local algorithm to intercept the new itemset discovery operation, such that itemset is thrown into one bucket by its own support value. Note that this change can be applied to any local FIM algorithm. After the local FIM algorithm finishes, all the itemsets are saved into memory, and organized in buckets. Then we sum up each bucket size one after another, and get a cumulative bucket size distribution. We do a simple linear regression analysis for it. From each bucket, we get a R-Square [22] value (R-Square value, which is between 0 and 1, measures how linear the data are in linear regression, the closer to 1, the more linear the data are. We get this value by using Apache statistics library [23]). Whenever it is below *R_Square_threshold*, we stop here and save the previous buckets' itemsets into HDFS. The intuition is that we think the bucket size is starting to grow non-linearly. To sum up, SmartCache has three parameters: *slack*, *buckets*, *R_Square_threshold*. Based on our experience, we propose their values to be 0.7, 20, and 0.8, respectively (please refer to Section V-B3 for sensitivity studies on these parameters).

Figure 6 shows one practical example when SmartCache's online analyzer is working. Each bar corresponds to one bucket, and there are 20 buckets in total. The ith bar's height means the sum of bucket's itemsets number from the 1st bucket to the ith bucket. Each bar corresponds to one R-Square value, it falls below 0.8 in bucket 13, so we put itemsets from bucket 1 to 12 into HDFS, and leave itemsets from bucket 13 to 20 behind. From this figure we can see that, starting from bucket 13, the cumulative size grows non-linearly. It means that our algorithm correctly detected the turning point for number of itemsets, or the turning point for I/O overhead.

Algorithm 3 and algorithm 4 have summarized SmartCache idea in pseudo code. $p1minsup$ represents phase 1's minimum support. In phase 1, we modify the local FIM algorithm such that whenever a new frequent itemset (relative to $p1minsup$) is discovered, it is either stored into L if its support is bigger than or equal to min_sup, or thrown into a bucket based on its support value (line 3 in alg. 3) if its support is between min_sup and $p1minsup$. After the local FIM algorithm finishes, we calculate the cumulative size distribution for the buckets (line 4 to 7 in alg. 3). Then we do the linear regression analysis to find the turning point (line 8 to 15 in alg. 3), and write specified itemsets' counting information to HDFS cache file (line 16,18,19 and line 20 to 24 in alg. 3). In phase 2, a candidate itemset is checked in cache first, the expensive scanning operation is only needed when it is not in cache. In the whole process, we do not change the data shuffled from phase 1's Map task to Reduce task (line 16,17,19 in alg. 3), so the global candidate itemsets remain the same with MRApriori.

Figure 7 shows the end-to-end time improvement of Zahra et al. solution and SmartCache, and the optimal point from our tuning. We can see from this figure that Zahra et al. solution can get 76.3% reduction in execution time compared with MRApriori, while SmartCache can get 91.8% reduction. Compared with Zahra et al. solution, SmartCache can get

Algorithm 3 SmartCache - Phase 1

```
1: procedure MAP(split, min_sup)
2:     p1minsup ← min_sup * slack
3:     L, B[1, buckets] ← FPGROWTH′(split, p1minsup)
4:     S[0] ← 0
5:     for i ← 1, buckets do
6:         S[i] ← S[i − 1]+ SIZE(B[i])
7:     end for
8:     for i ← 1, buckets do
9:         ADDDATA(i, S[i])
10:        r ← GETRSQUARE()
11:        if r < R_Square_threshold then
12:            limit ← i − 1
13:            break
14:        end if
15:    end for
16:    for all ⟨itemset, abs_sup⟩ ∈ L do
17:        EMITINTERMEDIATE(itemset, abs_sup)
18:        WRITE(itemset, abs_sup)
19:    end for
20:    for i ← 1, limit do
21:        for all ⟨itemset, abs_sup⟩ ∈ B[i] do
22:            WRITE(itemset, abs_sup)
23:        end for
24:    end for
25: end procedure

26: procedure REDUCE(itemset, list abs_sups)
27:    ▷ Omitted, the same with MRApriori
28: end procedure
```

Algorithm 4 SmartCache - Phase 2

```
1: procedure MAP(split, candidates)
2:     cache ← READ()
3:     for all itemset ∈ candidates do
4:         if itemset ∈ cache then
5:             abs_sup ← GET(cache, itemset)
6:             EMITINTERMEDIATE(itemset, abs_sup)
7:             continue
8:         end if
9:         abs_sup ← SCANCOUNT(itemset, split)
10:        EMITINTERMEDIATE(itemset, abs_sup)
11:    end for
12: end procedure

13: procedure REDUCE(itemset, list abs_sups, min_sup,
    total_rows)
14:    ▷ Omitted, the same with MRApriori
15: end procedure
```

65.4% reduction in execution time. Besides, SmartCache is close to the optimal point (the lowest point in Figure 4), in this example SmartCache takes 15.9% more execution time than the optimal point.

Note that the figures' statistics in Section III and Section IV are from a real workload $\langle pumsb, 85\% \rangle$. More experiment results are provided in Section V.

V. EVALUATION

The end of Section IV shows SmartCache's improvement on one dataset with one minimum support $\langle pumsb, 85\% \rangle$, this method is also effective for other $\langle dataset, minimum\ support \rangle$ combinations. In this section, we describe our experiment setup and present more of our experiment results[4].

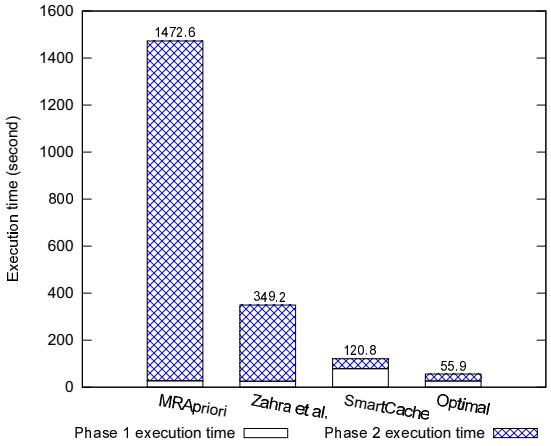


Fig. 7: SmartCache's improvement compared with MRApriori and Zahra et al. solution. Inputs are $\langle pumsb, 85\% \rangle$.

A. Experiment setup

Our testbed has three 64-bit virtual machines with the identical configuration. Each virtual machine has two VCPUs, 12GB memory, and uses 2.6.32 version Linux kernel. We use Hadoop version 2.2.0, which is based on YARN [24] architecture. We use replication ratio 3 for HDFS. Speculation for Map task and Reduce task is disabled. Since we want to make sure that two Map Tasks are not scheduled to the same node, each node is configured in a way that only one YARN container can run in it. We achieve this by setting YARN NodeManager's memory size to be 8GB, and container's minimum size also to be 8GB. Each task inside the container uses all the 8GB memory. We use 2 Map tasks and 1 Reduce task, because YARN ApplicationMaster has to use one container. Both phases do not use Reduce function as Combine function in the Map task side, because it is not necessary for phase 1, and correctness in phase 2 will be compromised if used. Before starting the experiment, we divide the input dataset into two splits with roughly equal number of rows, if the total row number is odd, then one split has one more row than the other. The purpose of this division is to try to minimize the load unbalance factor's influence. In practice, SmartCache does not need any assumption about the number of rows inside each split. Each execution time shown in this paper is the average of five runs.

As mentioned in Section IV, SmartCache has three tunable parameters *slack*, *buckets*, *R_Square_threshold*. Based on our experience, we propose them to be 0.7, 20, and 0.8. Even though when R-Square value falls down below 0.95, we can say the data are already not linear, we choose 0.8 as *R_Square_threshold* because we want to save as many itemsets as possible for saving scanning overhead, but to avoid saving huge amount of itemsets at the same time, in other words, to strike a balance between saving computation and reducing I/O overhead.

[4]Figure 1,2,3,4 in Section III and Figure 6,7 in Section IV are also got under the same experiment setup.

	pumsb	*accidents*	*T40I10D100K*	*kosarak*
Number of rows	49,046	340,183	100,000	990,002
Number of distinct items	2,113	468	942	41,270
Average number of items per row	74	33.8	39.6	8.1
Item's min ID	0	1	0	1
Item's max ID	7,116	468	999	41,270
Studied minimum support	90%, 85%	80%, 60%	1.3%, 1%, 0.7%	0.265%, 0.205%

TABLE I: Datasets' basic characteristics and minimum support values we have studied.

Four datasets *pumsb*, *accidents*, *T40I10D100K*, *kosarak* are chosen from an online public repository [13], which is specially designed for Frequent Itemset Mining research. Table I shows these four datasets' basic characteristics and the minimum support values we have studied. We can see that these four datasets' rows are from almost 50,000 to above 990,000; distinct items numbers are from almost 500 to above 40,000; average number of items per row varies from 8.1 to 74.

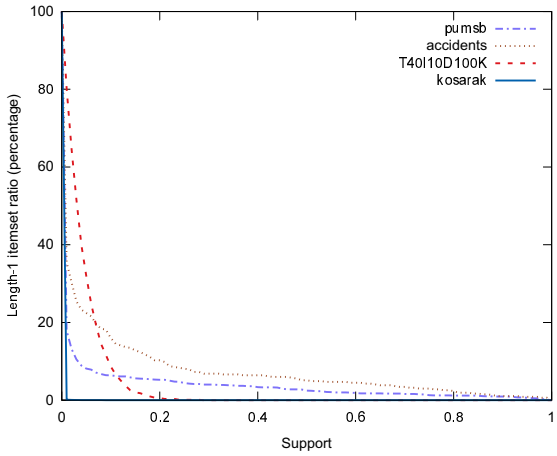


Fig. 8: Datasets' density analysis. Each point $d(x, y)$ in the figure means that in all length-1 itemsets, there are $y\%$ of them whose support values are bigger than or equal to x.

Even though there are some existing works that measure the density of datasets for FIM, and the four datasets we are using are classified into different groups in [25], there is no widely used formal way to define density in literature. Figure 8 provides our perspective for estimating dataset density. Every dataset has a non-increasing line in the figure. Each point $d(x, y)$ on the line means that in all length-1 itemsets, there are $y\%$ of them whose support values are bigger than or equal to x. We use length-1 itemset ratio as a density criterion for the reason that the higher length-1 itemset ratio, the more combinations of itemsets will likely be generated, the more dense the dataset. We can see that *kosarak* is the most sparse dataset, and all four datasets have different types of density.

Other than Zahra et al. solution, we have also compared SmartCache with BigFIM [17], [26]. BigFIM is deployed into Apache Mahout, which also uses Hadoop version 2.2.0 as the infrastructure. We set the number of Map tasks to be 2, and the depth of prefix tree to 3, as recommended in its documentation.

Because fundamentally Apriori is a pluggable component, and FP-Growth is faster than Apriori based on our experience, we replace Apriori by FP-Growth (our FP-Growth implementation is adapted from FPGrowth_itemsets algorithm from [27]) when implementing SmartCache for our experiment convenience. In this paper, we still call the baseline MRApriori, but the baseline's local FIM algorithm is also replaced by FP-Growth for a fair comparison.

B. Experiment results

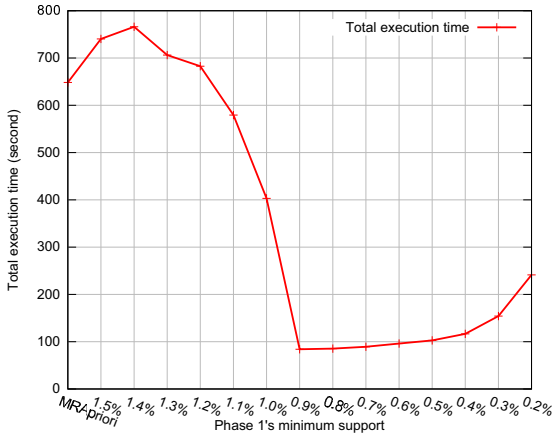
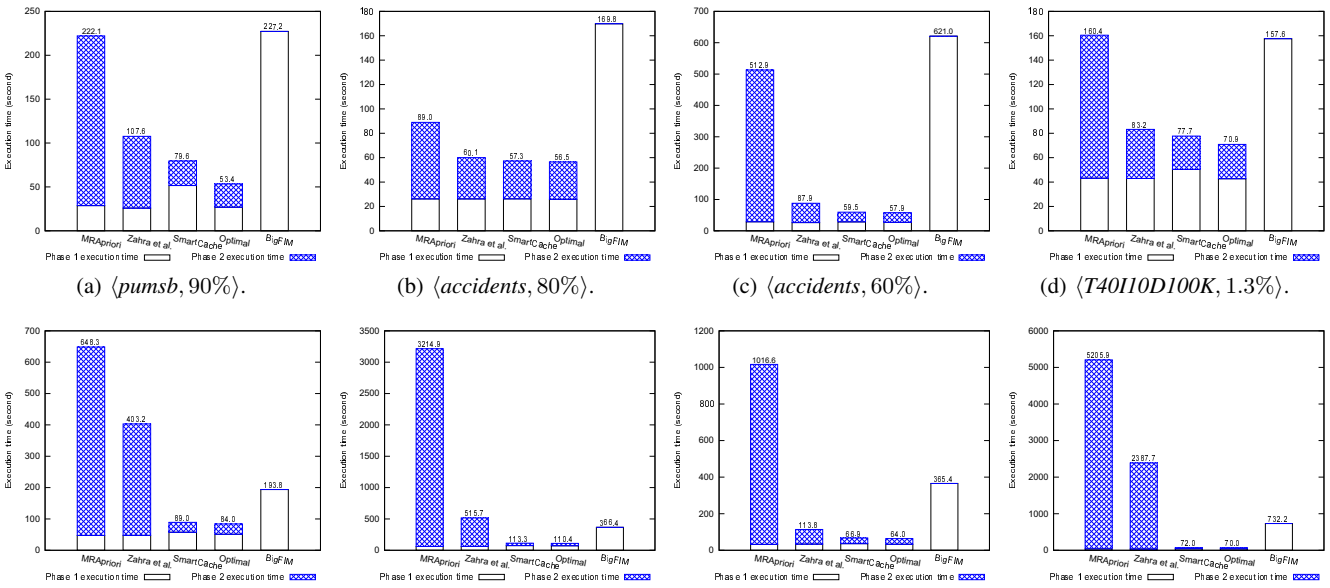


Fig. 9: Execution time as a function of phase 1's minimum support. Inputs are $\langle$*T40I10D100K*, $1\%\rangle$.

1) Overall Results: Every $\langle$*dataset*, *minimum support*$\rangle$ combination we studied has a similar performance curve to Figure 4. We use one more example to support this statement. From Figure 9 we can learn that in dataset *T40I10D100K* and minimum support 1%, the total execution time is also affected by phase 1's minimum support. The performance first becomes better, reaches its optimal point at 0.9%, and then becomes worse. This indicates that every input combination in our study needs an online algorithm to find the optimal point.

Figure 10a to 10h show all the input combination's performance comparison results except $\langle$*pumsb*, $85\%\rangle$, which was already discussed in Section III and Section IV. SmartCache is effective in all these scenarios. Compared with MRApriori, SmartCache reduces execution time by from 35.6% to 98.6%, on average 76.8%. This is because we add a cache layer to save scanning overhead in phase 2. Compared with Zahra et al. solution, SmartCache reduces execution time by from 4.6% to 97.0%, on average 45.4%. This is because our key observation is taking effect. Compared with BigFIM, SmartCache reduces execution time by from 50.7% to 90.4%, on average 70.9%. This is partly because SmartCache only uses two MapReduce phases, while BigFIM uses more than two. In the recommended configuration, BigFIM uses five MapReduce phases. Compared with the optimal point, SmartCache takes from 1.4% to 49.1% more time, on average 9.95%.

Figure 10a shows the experiment results when the input dataset is *pumsb* and minimum support is 90%. Zahra et al. solution reduces 51.5% execution time compared with MRApriori, while SmartCache reduces 64.1% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 26.0% execution time. SmartCache



(a) $\langle pumsb, 90\% \rangle$. (b) $\langle accidents, 80\% \rangle$. (c) $\langle accidents, 60\% \rangle$. (d) $\langle T40I10D100K, 1.3\% \rangle$.

(e) $\langle T40I10D100K, 1\% \rangle$. (f) $\langle T40I10D100K, 0.7\% \rangle$. (g) $\langle kosarak, 0.265\% \rangle$. (h) $\langle kosarak, 0.205\% \rangle$.

Fig. 10: Execution time comparison for different datasets and minimum support values.

takes 49.1% more time than optimal point, and compared with BigFIM, SmartCache reduces 65.0% execution time.

Figure 10b shows the experiment results when the input dataset is *accidents* and minimum support is 80%. Zahra et al. solution reduces 32.5% execution time compared with MRApriori, while SmartCache reduces 35.6% execution time compared with MRApriroi. Compared with Zahra et al. solution, SmartCache reduces 4.6% execution time. SmartCache takes only 1.4% more time than optimal point, and compared with BigFIM, SmartCache reduces 66.3% execution time.

Figure 10c shows the experiment results when the input dataset is *accidents* and minimum support is 60%. Zahra et al. solution reduces 82.9% execution time compared with MRApriori, while SmartCache reduces 88.4% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 32.3% execution time. SmartCache takes only 2.8% more time than optimal point, and compared with BigFIM, SmartCache reduces 90.4% execution time.

Figure 10d shows the experiment results when the input dataset is *T40I10D100K* and minimum suport is 1.3%. Zahra et al. solution reduces 48.1% execution time compared with MRApriori, while SmartCache reduces 51.6% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 6.6% execution time. SmartCache takes only 9.7% more time than optimal point, and compared with BigFIM, SmartCache reduces 50.7% execution time.

Figure 10e shows the experiment results when the input dataset is *T40I10D100K* and minimum support is 1%. Zahra et al. solution reduces 37.8% execution time compared with MRApriori, while SmartCache reduces 86.3% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 77.9% execution time. SmartCache takes only 6.0% more time than optimal point, and compared with BigFIM, SmartCache reduces 54.1% execution time.

Figure 10f shows the experiment results when the input dataset is *T40I10D100K* and minimum support is 0.7%. Zahra et al. solution reduces 84.0% execution time compared with MRApriori, while SmartCache reduces 96.5% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 78.0% execution time. SmartCache takes only 2.6% more time than optimal point, and compared with BigFIM, SmartCache reduces 69.1% execution time.

Figure 10g shows the experiment results when the input dataset is *kosarak* and minimum support is 0.265%. Zahra et al. solution reduces 88.8% execution time compared with MRApriori, while SmartCache reduces 93.4% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 41.2% execution time. SmartCache takes only 4.6% more time than optimal point, and compared with BigFIM, SmartCache reduces 81.7% execution time.

Figure 10h shows the experiment results when the input dataset is *kosarak* and minimum support is 0.205%. Zahra et al. solution reduces 54.1% execution time compared with MRApriori, while SmartCache reduces 98.6% execution time compared with MRApriori. Compared with Zahra et al. solution, SmartCache reduces 97.0% execution time. SmartCache takes only 3.4% more time than optimal point, and compared with BigFIM, SmartCache reduces 90.2% execution time.

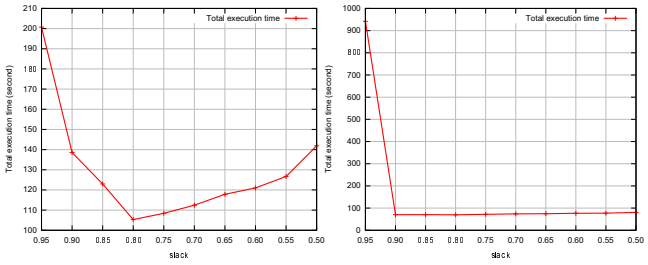
2) Runtime Statistical Results: Table II shows three metrics while running SmartCache in different workloads. Phase 1 final minimum support shows the turning point SmartCache chose after its online analysis. Size of saved itemsets shows the cache size. Phase 2 scanning avoidance ratio shows the effect of this cache. Each cell has two values because there are two Map tasks in our testbed. From Table II's second column, we can see that SmartCache can effectively make its decision indepen-

	phase 1 final minimum support	phase 1 size of saved itemsets	phase 2 scanning avoidance ratio
$\langle pumsb, 90\% \rangle$	76.5%, 77.85%	20MB, 14MB	100%, 100%
$\langle pumsb, 85\% \rangle$	69.7%, 69.7%	176MB, 147MB	100%, 100%
$\langle accidents, 80\% \rangle$	56%, 56%	96KB, 92KB	100%, 100%
$\langle accidents, 60\% \rangle$	42%, 42%	728KB, 696KB	100%, 100%
$\langle T40I10D100K, 1.3\% \rangle$	1.027%, 0.988%	1.4MB, 1.8MB	100%, 100%
$\langle T40I10D100K, 1\% \rangle$	0.7%, 0.7%	22MB, 24MB	100%, 100%
$\langle T40I10D100K, 0.7\% \rangle$	0.49%, 0.49%	51MB, 52MB	100%, 100%
$\langle kosarak, 0.265\% \rangle$	0.2014%, 0.2173%	664KB, 524KB	100%, 100%
$\langle kosarak, 0.205\% \rangle$	0.1435%, 0.1435%	8.8MB, 13MB	100%, 100%

TABLE II: Workload metrics after using SmartCache.

dently based on that split's specific characteristics. Table II's fourth column shows that in these workloads, SmartCache can achieve 100% scanning avoidance ratio.

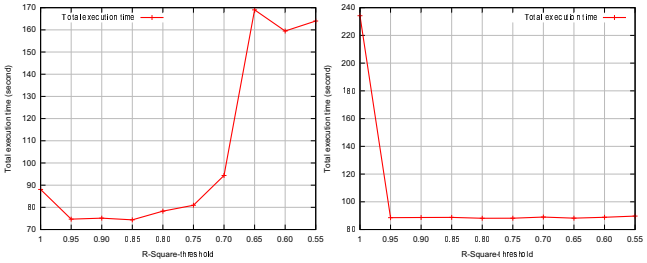
SmartCache has a low execution time overhead. From Figure 10b, 10c, 10d, and 10g, we can see that even though the Zahra et al. solution is already close to optimal point (within twice of optimal point), SmartCache is still between Zahra et al. solution and optimal point. SmartCache also has a low space overhead. Table II's third column shows that the maximum cache size is only less than 200MB. Studying cache size under other datasets and other minimum support values is for our future work.



(a) $\langle T40I10D100K, 0.7\% \rangle$. (b) $\langle kosarak, 0.205\% \rangle$.

Fig. 11: *slack* parameter sensitivity study for two datasets and minimum support values.

3) Sensitivity Study: Figure 11 shows our sensitivity study result for *slack* parameter in SmartCache. *slack* parameter is responsible for lowering phase 1's minimum support such that more itemsets can be considered to be saved into cache or not. *slack* parameter also decides the maximum amount of itemsets that could possibly be saved into cache. We change *slack* parameter from 0.95 to 0.50 and decrease it by 0.05 at a time. From Figure 11a we can see that the execution time is at its highest point when *slack* is 0.95. This is because some itemsets that could contribute to a higher scanning avoidance ratio have not been saved into cache. The above reason also explains the Figure 11b's highest point. In Figure 11a, execution time reaches its lowest point when *slack* is lowered to 0.80, and gets worse afterwards. The execution time increase happens because of redundant computation and extra I/O overhead. For Figure 11b, execution time remains stable when *slack* drops below 0.90. This is because for this specific dataset, lowering phase 1's minimum support to half of minimum support does not introduce a substantial number of itemsets into cache.



(a) $\langle pumsb, 90\% \rangle$. (b) $\langle T40I10D100K, 1\% \rangle$.

Fig. 12: *R_Square_threshold* parameter sensitivity study for two datasets and minimum support values.

Figure 12 shows our sensitivity study result for *R_Square_threshold* parameter in SmartCache. *R_Square_threshold* parameter is responsible for finding the turning point after which the number of itemsets start to grow non-linearly. We change *R_Square_threshold* parameter from 1 to 0.55 and decrease it by 0.05 at a time. From Figure 12a we can see that the execution time is not at lowest point when *R_Square_threshold* is 1. This is because scanning avoidance ratio did not reach its peak point. The above reason also explains the Figure 12b's highest point. In Figure 12a, execution time reaches it lowest point when *R_Square_threshold* is lowered to 0.85, and gets worse afterwards. The execution time increase happens because unnecessary large amount of itemsets have been introduced into cache. For Figure 12b, execution time remains stable when *R_Square_threshold* drops below 0.95. This is because for this specific dataset, within the *slack range*, the cumulative bucket size did not increase significantly across buckets.

VI. Discussion

Our key observation mentioned in Section III leads to two changes for the local FIM algorithm. First, minimum support parameter is lowered. This means that local FIM algorithm needs extra memory to hold the corresponding itemsets. In production run, we can take the physical memory size limitation into consideration, and tune the *slack* parameter. Second, local FIM's frequent itemset discovery operation is intercepted. This interception is independent of local FIM algorithm's main logic, so we did not destroy the advantage of two-phase MapReduce FIM algorithm: we automatically benefit from a better local FIM algorithm.

The key observation itself can be applied to other frequency mining algorithm in MapReduce as well, such as Frequent Tree Mining and Frequent Graph Mining [28]. Even though all the experiments are done in a three node cluster, we believe the key observation's effectiveness is independent of the cluster size. Trying SmartCache in a larger cluster, which invovles adjusting the three parameters, is for our future work.

After Hadoop, a large amount of large data processing frameworks are developed. Some of them can handle large graphs, such as Pregel [29], GraphLab [30]. Some of them can handle large data streams, such as MapReduce Online [31], Yahoo! S4 [32], Twitter Storm [33]. Recently, a memory-based large data processing framework Spark [34] and Spark Streaming [35], which stores intermediate results in memory, is

developed. Our solution can guide the design in Spark because not all intermediate counting information can be cached.

VII. Conclusion

Researchers have tried to apply the traditional data mining topic Frequent Itemset Mining into MapReduce framework to meet the big data challenge. However, the two-phase MapReduce FIM algorithm, which is the most advanced way so far, suffers from high scanning overhead problem. We identified new improvement space on top of the state-of-the-art solution, and proposed a new regression based method to determine the optimal size of cache. Extensive experiments had been done on four public datasets. Our solution SmartCache can reduce on average 45.4%, and up to 97.0% execution time compared with the state-of-the-art solution.

Acknowledgments

The authors would like to thank the anonymous reviewers and Mai Zheng for their helpful feedback. This research is partially supported by NSF grants #CCF-0953759 (CAREER Award) and #CCF-1319705.

References

[1] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

[3] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns Without Candidate Generation," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 1–12.

[4] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New Algorithms for Fast Discovery of Association Rules," in *In 3rd Intl. Conf. on Knowledge Discovery and Data Mining*. AAAI Press, 1997, pp. 283–286.

[5] "Amazon Was Selling 306 Items Every Second At Its Peak This Year," http://www.businessinsider.com/amazon-holiday-facts-2012-12.

[6] "Taobao Total Sales Reached USD 5.7 Billion on One Single Day," http://www.chinainternetwatch.com/4691/taobao-bachelors-day/.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[8] "Apache Hadoop," http://hadoop.apache.org/.

[9] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based Frequent Itemset Mining Algorithms on MapReduce," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ser. ICUIMC '12. New York, NY, USA: ACM, 2012, pp. 76:1–76:8.

[10] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel Implementation of Apriori Algorithm Based on MapReduce," in *Proceedings of the 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. SNPD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 236–241.

[11] O. Yahya, O. Hegazy, and E. Ezat, "An efficient implementation of Apriori algorithm based on Hadoop-MapReduce model," *International Journal of Reviews in Computing*, vol. 12, pp. 59–67, 12 2012.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[13] "Frequent Itemset Mining Dataset Repository," http://fimi.ua.ac.be/data/.

[14] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-Growth for Query Recommendation," in *Proceedings of the 2008 ACM Conference on Recommender Systems*, ser. RecSys '08. New York, NY, USA: ACM, 2008, pp. 107–114.

[15] "Apache Mahout," https://mahout.apache.org/.

[16] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel FP-Growth with MapReduce," in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, Nov 2010, pp. 243–246.

[17] S. Moens, E. Aksehirli, and B. Goethals, "Frequent Itemset Mining for Big Data," in *Big Data, 2013 IEEE International Conference on*, Oct 2013, pp. 111–118.

[18] L. Li and M. Zhang, "The Strategy of Mining Association Rule Based on Cloud Computing," in *Proceedings of the 2011 International Conference on Business Computing and Global Informatization*, ser. BCGIN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 475–478.

[19] L. Wang, L. Feng, and P. L. Jing Zhang, "An Efficient Algorithm of Frequent Itemsets Mining Based on MapReduce," *Journal of Information and Computational Science*, vol. 11, no. 8, pp. 2809–2816, 5 2014.

[20] Z. Farzanyar and N. Cercone, "Efficient Mining of Frequent Itemsets in Social Network Data Based on MapReduce Framework," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '13. New York, NY, USA: ACM, 2013, pp. 1183–1188.

[21] ——, "Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce-Based Approach," in *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, Dec. 2013, pp. 592–598.

[22] "R-Square in Wikipedia," http://en.wikipedia.org/wiki/R-square.

[23] "Apache Statistics Library," http://commons.apache.org/proper/commons-math/userguide/stat.html.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[25] F. Flouvat, F. De Marchi, and J.-M. Petit, "A new classification of datasets for frequent itemsets," *Journal of Intelligent Information Systems*, vol. 34, no. 1, pp. 1–19, 2010.

[26] "BigFIM project," https://gitlab.com/adrem/bigfim/tree/master.

[27] "SPMF, An Open-Source Data Mining Library," http://www.philippe-fournier-viger.com/spmf/.

[28] Y. Wang, S. Parthasarathy, and P. Sadayappan, "Stratification driven placement of complex data: A framework for distributed data analytics," in *ICDE*, 2013, pp. 709–720.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[30] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[31] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[32] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177.

[33] "Twitter Storm," https://storm.incubator.apache.org/.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[35] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438.