MapReduce SON Project

CMPT 741

Colin J. Brown : 301185603
Jeremy Kawahara : 301026214

Table of Contents

Notation:

s = user supplied support threshold (percentage)

k = number of sub-files (integer)

n = number of lines within a (sub-)file (i.e., number of baskets)

z = size of an itemset

Description of the project

Our project goal was to implement the two pass MapReduce algorithm of Savasere, Omiecinski, and Navathe, known as the SON algorithm after the authors. This SON algorithm, as outlined in the assignment and the course textbook, has two main MapReduce phases. In the *first phase* of MapReduce, we break a large datafile down into smaller sub-files such that

each sub-file is small enough that we can find all the frequent itemsets in memory. Each sub-file is passed to a mapper, which should ideally be on its own machine. The mapper takes the sub-file and finds the frequent itemsets within the sub-file (using an approach like the Apriori algorithm). This *first-phase-mapper* then passes the frequent itemsets to the *first-phase-reducer*. The *first-phase-reducer* then groups all the frequent itemsets to output unique frequent itemsets that were found to be frequent in at least one of the sub-files. This list of itemsets is our *candidate itemsets* that serves as input to the *second phase*.

In the *second phase* we again run MapReduce, except this time we have a list of *candidate itemsets* and want to check if they are *frequent itemsets* when considering the entire dataset. To do this, the *second-phase-mapper* takes as input one basket at a time and checks to see if it contains any itemsets that are in the *candidate set*. If an itemset is within the *candidate itemsets*, the itemset's frequency is computed and both the itemset and frequency are passed to the *second-phase-reducer*. Within the *second-phase-reducer*, we group and count the number of unique itemsets within the entire dataset. We then compare if the itemset is frequent by comparing the frequency of the itemset to the user specified threshold. Itemsets that are found to be greater than or equal to the user specified threshold are then accepted as *frequent itemsets*.

High level approach to the key steps

Here we describe in more detail each of the key steps to our approach. Our implementation of SON proceeds in four main steps: 1) input splitting, 2) candidate search, 3) candidate filtering and 4) output formatting.

We include this diagram as a visual outline of our approach.

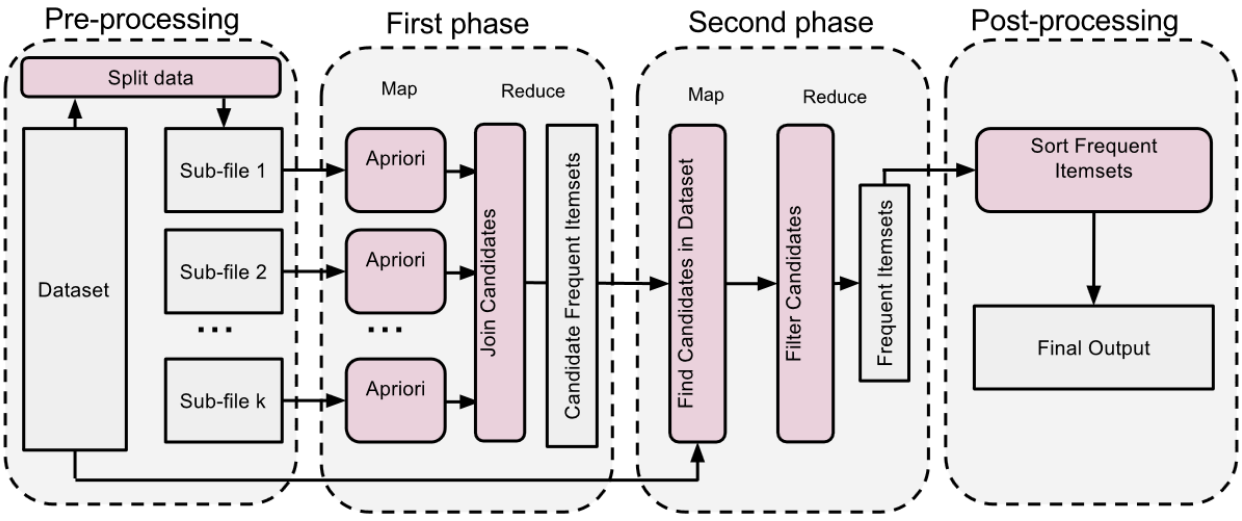


Fig. 1 - High level approach to our implementation of the SON MapReduce Algorithm

Input Splitting

Hadoop offers no way to force a specific number of map tasks to be generated. Instead, it dynamically chooses a number of mappers appropriate to the input. Typically, Hadoop will generate one mapper per input file. It may generate more if an input file is larger than the specified block size. Thus, in order to split our task of finding candidates into k chunks, we first split our input file into k sub-files. This unfortunately requires an extra reading pass of our dataset and one more pass writing everything out into the split files, turning a 2-pass method into a 4-pass method.

Candidate Search (First Phase of SON)

This step finds the *candidate itemsets* by examining subsets of the data and aggregating itemsets. Itemsets who are found to be frequent within a sub-file, we call *candidate itemsets*.

Mapper

For each mapper, the SON algorithm performs the Apriori algorithm entirely in memory over $1/k$ of the original input to find a set of candidate itemsets. Thus, for each of the k mappers, we load the entire associated file chunk into memory at once. This poses a challenge in Hadoop since there is no standard *Input Format* which provides an entire file as the value in the key-value pair. By default, the mapper will only read in a single line (i.e., a single basket) at a time. However, if we only load a single line, then this effectively changes the k to be equal to all the number of lines in the file. This would have the effect of making all itemsets being supported. Thus no itemsets would be filtered in the first pass. This required us to overwrite the default behaviour of the mapper by writing our own *Input Format* and *Record Reader* (*FileChunkInputFormat.java*, and *FileChunkRecordReader.java*, respectively) which provide each mapper with a single key-value pair containing the last position in the file and the *entire contents of the sub-file*, respectively.

Since we are working within a sub-file, we have to adjust the user-specified threshold s to account for the smaller file size. We compute the local support threshold, s_{local} as follows,

$$s_{local} = s \times n_{sub} = \frac{s \times n}{k}$$

where s is the user supplied support threshold, n_{sub} is the number of lines within a sub-file, and k is the number subfiles as specified by the user.

Once the file is loaded into memory, each mapper performs an in-memory version of the Apriori algorithm to find all itemsets with frequency greater than one kth of the original support threshold. At a high level, the Apriori algorithm works by finding itemsets of increasing size, z. Once supported itemsets of size z have been found, all items not used by those itemsets are removed from the data. In this way, it leverages the concept of monotonicity: all

subsets of frequent itemsets must also be frequent. The Apriori algorithm uses the computed s_{local} as the threshold to determine which itemsets are frequent.

The output of this mapper is the *candidate itemsets* within this subset of data. This map step can be described in pseudocode as follows:

```
MAPPER(key, value)
% key : last position in a file (not used)
% value : a subset of baskets separated by a newline character

baskets = SPLIT_BY_NEWLINE(value)
candidate_frequent_itemesets = APRIORI(baskets, s_local)
FOR EACH itemset IN candidate_frequent_itemsets
     OUTPUT(itemset, 1)
```

Reducer

Here we aggregate the *candidate itemsets* found from the previous mapper to output the *unique candidate itemsets* for the second phase. As we are going to compute the true frequencies of the itemsets in the second phase, we do not need to sum the frequencies here (although we do so in the actual code for debugging purposes).

```
REDUCER (key, values)
% key : itemset
% values : frequencies of all found copies of the itemset

OUTPUT(key, sum(values)) %Output sum(value) only for debugging
```

The output is our set of *candidate itemsets* found within all the sub-files.

Candidate Filtering (Second Phase of SON)

In this phase we assume that the *candidate frequent itemsets* have already been computed. The main idea is that the *candidate itemsets* computed in the first phase will contain all the frequent itemsets but may also contain itemsets that are not frequent when we consider the entire dataset. Thus we are removing the candidates from the first phase that are *false positives*.

Setup

Prior to calling the mappers, we first load the *candidate itemsets* found from the first phase into memory. This loads all the *candidate itemsets* into the memory of all the mappers in the form of a *HashSet* to allow for fast access. The frequency found in the first phase is ignored as we will now compute the frequency for the entire dataset.

Mapper

Each mapper now has access to the *candidate itemsets* computed in the first phase. As input, the value is a single basket (i.e., a line from the file). Starting from itemsets of size 1 (i.e., a single integer in the list), we check if each item is in the *candidate itemsets*. If the singleton item is a candidate, then we output the singleton as the key along with a count of 1 as a the value. Similar to the Apriori algorithm, we use monotonicity here to find candidate subsets efficiently. For a given subset size, all items in a basket that are not part of any candidates are removed from the basket. Thus, there are fewer items to search over in subsequent passes.

```
MAPPER(key, value)
% key : line number
% value : basket

basket = value
candidates = GET_CANDIDATES() % loaded in the setup
FOR itemSize=1 TO LENGTH(basket)
     [candidates, usedItems] = ADD_CANDIDATES(itemsize, basket,
candidates)
     basket = REMOVE_NON_FREQUENT_ITEMS(basket, usedItems)

FOR EACH itemset IN candidates
     OUTPUT(itemset, 1)
```

Reducer

The reducer's *key* is the *candidate itemset* and the *value* is the itemset's frequency (i.e., the number of times it appeared in the dataset). The reducers group all the itemsets by key so this value will be the frequency, $f_{itemset}$, in the entire dataset. We now check if this frequency, $f_{itemset}$, is greater than or equal to the user supplied support threshold s. Since s is given as a percentage, we multiply s by the total number of lines n appearing in our dataset, to give us the number of times the itemset must appear to be considered frequent. Thus if,

$$f_{itemset} \geq s \times n$$

then we accept this itemset as frequent and output the itemset and frequency as the key value pair, respectively.

```
REDUCER(key, values)
% key : itemset
% values: list of frequencies of the itemset
s = GET_SUPPORT() % Support threshold specified by the user.
n = GET_NUM_OF_LINES() % Number of lines in dataset (computed once).

freq = 0
FOR value IN values
     freq = freq + value

IF freq >= (s × n)
     OUTPUT(itemset, freq)
```

Postprocessing

The *key value* output from the final reducer is not in the desired output format as specified in the project details so we modify it as follows: We count the number of entries in the frequent itemsets file and output how many frequent itemsets were found. We then load the *frequent itemsets* into memory and sort them from most frequent to least frequent. Each itemset is output on one line along with its count.

Assumptions

In implementing the SON algorithm, we made the following assumptions:

1) We assume that we don't have to choose a random subset of the file as discussed in the text. There is no "p" value to pass to our method so we assume that we use all our data in the subfiles. This would be relatively easy to modify in our code where we would just add a line to the subfile in the *Input Splitting*-phase based on a user specified probability "p".

2) We assume that the items in each line (i.e. basket) of the input text has already been sorted in order as was done in the *example.dat*.

Sample test data and results

To test our method we prepared a very small test case which we could compute the expected result by hand. Here we include our test case, expected result and actual output.

Test data

We use 8 baskets with 4 items {1,2,3,4}. Each line represents a basket composed of items separated by whitespaces.

1 2
1 3
2 3
1 2 3
1 2 3
4
1 2 3
1 2 3 4

Test 1 - Filesplitter check

Here we check if the files are being split up correctly. We set $k = 2$, meaning that we split the file into two equal sized chunks with 4 items in each. Thus we expect two files containing items:

File 1	File 2
1 2 1 3 2 3 1 2 3	1 2 3 4 1 2 3 1 2 3 4

We check the output files that are created in the *Input Splitting* phase and find that indeed there exists two sub-files composed of these baskets.

Test 2 - Candidate frequent itemset check

Here we check that the expected *candidate itemsets* are found in each mapper. We set $k = 2$, to split the sub-files as described above and set the support threshold $s = 5/8$. Each mapper receives n/k baskets, were n is the total number of lines in the file. Thus within a mapper we lower the local threshold to $s_{local} = s(n/k) = (5/8)(8/2) = 5/2 = 2.5$.

Thus for an itemset to be considered frequent, it must occur at least 3 times within the subfile. Thus we would expect the candidate itemsets to be as follows.

File 1 Candidates	File 2 Candidates
1 2 3	1 2 3 4 1 2

	1 3 2 3 1 2 3

In the *first-phase-reducer*, we expect these *candidate itemsets* to be grouped together as:
1
2
3
4
1 2
1 3
2 3
1 2 3

We check the output created by the *first-phase-reducer* and find that these same *candidate itemsets* are created.

Test 3 - Check entire pipeline

Here we try our entire method with different support thresholds *s*. We also specify different *k* values of *k* = {1,2,4} and verify that the results do not change.

itemset	Expected frequency with s=1/8	Actual frequency with s=1/8	Expected frequency with s=2/8	Actual frequency with s=2/8	Expected frequency with s=4/8	Actual frequency with s=4/8
1	6	6	6	6	6	6
2	6	6	6	6	6	6
3	6	6	6	6	6	6
4	2	2	2	2	-	-
1 2	5	5	5	5	5	5
1 3	5	5	5	5	5	5
1 4	1	1	-	-	-	-
2 3	5	5	5	5	5	5
2 4	1	1	-	-	-	-

3 4	1	1	-	-	-	-
1 2 3	4	4	4	4	4	4
1 2 4	1	1	-	-	-	-
1 3 4	1	1	-	-	-	
2 3 4	1	1	-	-	-	-
1 2 3 4	1	1	-	-	-	-

Test 4 - Compare results of Apriori using example.dat

Here we compare our implementation results with results from existing online code that executes a single in-memory Apriori algorithm over the entire dataset. We modified existing Apriori code from:
https://gist.github.com/monperrus/7157717

As well, we used an example of how to call it from the client here:
https://code.google.com/p/rockit/source/browse/trunk/rockit/src/main/java/com/googlecode/rockit/app/ExampleOfClientCodeOfApriori.java?r=160

We ran the existing Apriori code with a support of $s = 0.05$ and $s = 0.01$ over the entire 100,000 baskets found in *example.dat* and compared it to our implementation with the same thresholds using $k = 10$. The following itemsets were found to be frequent:

itemset $s = 0.05$	frequency $s = 0.05$	itemset $s = 0.01$ (pairs/triplets only shown)	itemset $s = 0.01$ (pairs/triplets only shown)
217	5375	39 704	1107
354	5835	789 829	1194
368	7828	368 829	1194
419	5057	227 390	1049
494	5102	39 825	1187
529	7057	217 346	1136
684	5408	368 682	1193
722	5845	704 825	1102

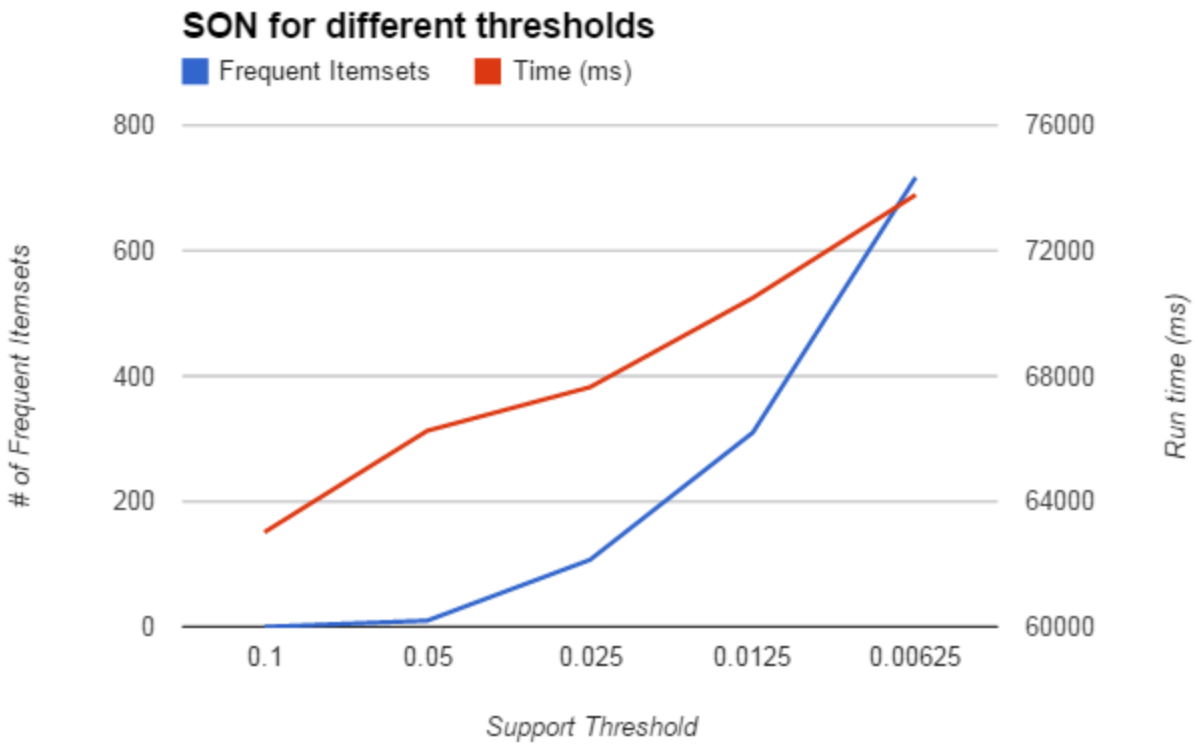
766	6265	390 722	1042
829	6810	39 704 825	1035

With $s = 0.01$ we found 375 frequent singletons, 9 frequent pairs, and 1 frequent triple.

We compared this to our implementation and found our results match.

Test 5 - Runtime vs # of frequent itemsets using example.dat

In our last test, we wanted to explore the number of frequent itemsets for different support thresholds and the runtime to find them using the SON algorithm. Using k=10 for all tests over the example.dat dataset, we found an exponential increase in number of frequent itemsets as we divided the support threshold in half (i.e. log linear decrease, so number of frequent itemsets is doubly exponential across support thresholds). In contrast, the runtime (computed as total time for all map and reduce tasks) increased roughly linearly.



Discussions

Our experiments demonstrated that our implementation of the SON algorithm was both correct and efficient, returning the expected itemsets in a runtime growing sub-linearly in the number of frequent itemsets recovered.

The most challenging part of this project was understanding how to convert the SON algorithm to hadoop code. In particular, it was not clear how to correctly get the mappers in the first phase to read the desired number of lines (based on k) into the main memory. By default, the mapper only reads a single basket (line within the text file) into memory at a time and there appears to exist no standard way to read more. Our first approach was to try to write an *input format* that would only read some chunk of a file. However, this would limit hadoop to only one mapper machine, reading chunks serially, rather than in parallel. Our next approach was to split the input into multiple files and then pass each mapper a filename of one of the parts. Unfortunately, this would mean that a mapper could be reading from a non-local file (i.e. network transfer). Even worse, this approach provides no way of specifying how many mappers hadoop should launch.

Another major challenge working with hadoop is that it is very difficult to debug code that is running simultaneously on multiple machines. We made heavy use of false ‘info’ outputs and exception trace stacks.

Appendix

Given that our code is well over a thousand lines long and spread over 7 files, we decided to not copy it here, but instead include our code files along with this document.