Implementation of PrefixSpan in Apache Flink

Jens Röwekamp, Tianlong Du

15th February 2016

Abstract

Data mining is commonly used by retail organizations to solve decision support problems and mining frequent items is usually among the first steps. In this paper, we show the implementation of a sequential pattern mining algorithm called PrefixSpan in Apache Flink and the experimental evaluation comparing with Apache Spark. The Flink implementation converts the recursive algorithmn to an iteration operation to execute parallel. The disk space needed for candidate patterns is the bottleneck of the approach and needs to be improved in the future.

1 Introduction

Retail organizations usually collect and store massive amounts of sales data. In one transaction date, customer-id and the items bought are recorded. Data mining can be used on massive data to solve decision support problems. If a customer bought something whilst online shopping, for example a SanDisk USB3.0 64GB pen drive, retailers would give a recommend, like "customers who bought this item also bought: ..." To give this recommend, the recommended items must be frequently bought together with the ones which the customer bought. This can be queried from the "frequently bought together items" database. To generate this database, we need to find out frequent patterns from the transaction database. Mining frequent items is usually among the first steps to analyze a large-scale dataset.
In this project, we implemented a sequential pattern mining algorithm called PrefixSpan [1] in Apache Flink [2] and compared its performance with a reference implementation in Apache Spark [3], which is another platform for the distributed processing of massive amounts of data.

Here we use the same definition as in the original paper. [1]

Let $I = \{i_1, i_2, \dots, i_n\}$ be a *set* of all **items**.

An **itemset** is a *subset* of items.

A **sequence** is an *ordered* list of itemsets. A sequence s is denoted by $\langle s_1 s_2 \cdots s_l \rangle$, where $\mathrm{s_j}$ is an itemset. $\mathrm{s_j}$ is also called an element of the sequence, and denoted as $(x_1 x_2 \cdots x_m)$, where $\mathrm{x_k}$ is an item. For brevity, the brackets are omitted if an element has only one item, i.e. element (x) is written as x.

The number of instances of items in a sequence is called the **length** of the sequence.

A **sequence database** S is a set of tuples $\langle$sid, s$\rangle$, where sid is a sequence_id and s a sequence.

The **support** of a sequence α in a sequence database S is the number of tuples in the database containing α.

Given a positive integer *min_support* as the support threshold, a sequence α is called a **sequential pattern** in sequence database S if $support_S(\alpha) \geq min\_support$.

Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n \rangle$, a sequence $\beta = \langle e'_1 e'_2 \cdots e'_m \rangle$ $(m \leq n)$ is called a **prefix** of α if and only if

1. $e'_i = e_i$ for $(i \leq m-1)$ and
2. $e'_m \subseteq e_m$ and
3. all the frequent items in $(e_m - e'_m)$ are alphabetically after those in e'_m.

For example, $\langle a \rangle, \langle aa \rangle, \langle a(ab) \rangle$ *and* $\langle a(abc) \rangle$ are prefixes of sequence $s = \langle a(abc)(ac)d(cf) \rangle$, but neither $\langle ab \rangle$ nor $\langle a(bc) \rangle$ is considered as a prefix if every item in the prefix $\langle a(abc) \rangle$ of sequence s is frequent in S.

Let β be be the prefix of α, the following part in the sequence after the prefix as sequence γ is called **suffix** with regards to prefix β. The relation of sequence α, prefix β and suffix γ is denoted as $\alpha = \beta \cdot \gamma$.

Let α be a sequential pattern in a sequence database S. The **α-projected database**, denoted as $S|_\alpha$, is the collection of suffixes of sequences in S with regards to prefix α.

Problem statement: Given a sequence database and the min_support threshold, sequential pattern mining is to find the complete set of sequential patterns in the database.

Our paper proceeds as follows: Firstly, the algorithm detail is represented. Then the implementation in Apache Flink is shown. Afterwards a case study is introduced to compare the performance with a reference implementation in Apache Spark. At last, the paper ends with a conclusion.

2 Algorithm detail

Given a sequence database, the first step is to fix the order in itemsets. Because items in an itemset can be listed in any order, one can fix the order of items within each itemset to avoid checking every possible combination of a potential candidate sequence. With such a convention the expression of a sequence is unique. For example, instead of $\langle a(bac)(ca)d(fc)\rangle$, the items can be listed alphabetically in itemsets as $\langle a(abc)(ac)d(cf)\rangle$.

Secondly, we start a first scan to find the length-1 frequent items. Infrequent items are filtered out because of the following property: A sequence whose prefix is an infrequent sequence can't be a frequent sequential pattern.

Next, we use database projection to do divide and conquer. According to the set of frequent items, the complete set of sequential patterns can be divided into disjoint subsets. For each basic frequent item used as a prefix, we make a projected sequence database and scan the projected database to find its length-1 frequent patterns. The new pattern either can be assembled to the last element of the prefix to form a length-2 sequential pattern ⟨ (ab) ⟩, or can be appended to the prefix to form a length-2 sequential pattern ⟨ ab ⟩. Recursively, the projected database can be further projected by the respective new patterns to find length-3, length-4, length-5, ..., patterns. So the length of frequent pattern grows by 1 after each scan. The mining process ends until no more frequent patterns are found.

The following pseudo-code shows the recursive mining process:

```
// pseudocode
PrefixSpan(prefix,sequence_db){
  // find new frequent pattern
  length_one_patterns=FindLengthOnePattern(
      sequence_db)
  // devide and conquer
  for(pattern in length_one_patterns){
    frequent_pattern=combine(prefix,pattern)
    // collect result
    output frequent_pattern
    // project new sequence database
    projected_db=sequence_db.project(pattern)
    PrefixSpan(frequent_pattern,projected_db)
  }
}
// to call it
PrefixSpan("", sequence_db)
```

The major cost of PrefixSpan is database projection. Pseudoprojection is used to improve performance. Instead of performing physical projection, one can register the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence. However, because random access disk space is costly, if the original sequence database or the projected databases is too large to fit into main memory (disk-based accessing situation), physical projection should be applied. Whenever the projected databases can fit in main memory, then use pseudoprojection.

3 Flink Implementation

As Flink in its current version 0.10.1 doesn't offer native mechanisms to execute recursive functions, we used two different approaches to implement the PrefixSpan algorithm. The first uses Java recursion, which calls the regarding Flink functions. The second transforms PrefixSpan's recursive divide and conquer approach to an iterative one and uses Flink's Delta Iteration to be executed natively.

Input Format

To implement the PrefixSpan algorithm we defined our sequence database format of a DataSet$\langle Tuple2\,\langle Long, int[\,]\rangle\rangle$ where the first entry is the sequence id and the second the regarding sequence. Zeros were used as itemset delimiter and needed to begin and end a sequence. To be able to use different input sources we defined an InputDecoder interface which is responsible for parsing the input source into our sequence database format. Furthermore we implemented four different Decoder classes which implement the InputDecoder interface and are used for both, the recursive and iterative, approaches.

The *DummyInputDecoder* defines three sample databases to test if the PrefixSpan algorithm is implemented correctly.

The *PlainInputDecoder* transforms comma separated input files of the format "sequence_id, 0, first item, second item, 0, third item, ..., nth item, 0" into the database format.

The *BigBenchInputDecoder* transforms BigBench's [4] store_sales.dat, which stores customer purchases, into the database format.

The *IBM_DataGeneratorInputDecoder* transforms customer purchase files generated by IBM's quest data generator [5] into the database format.

3.1 Recursive approach

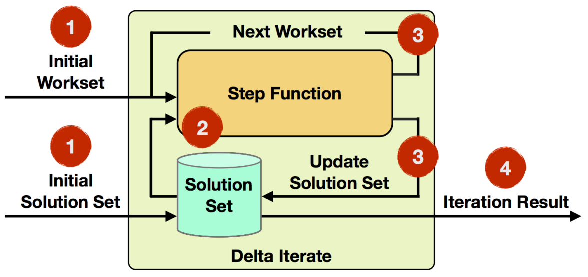
The recursive implementation of the PrefixSpan algorithm was mostly analogue to the pseudocode stated in section 2.

First all frequent length-1 items are detected by flatMapping the database, grouping by the prefix, summing up the support of each prefix and filtering the infrequent prefixes.
Afterwards all infrequent items were deleted from the database to reduce the further amount of projections.
In a third step the recursion was started to find all frequent sub-patterns for each frequent length-1 item. Here MapReduce techniques and sequence position pointers were used to calculate the support of each item.
In the end the results were collected, sorted and written to a csv file for further access.

The source code of this implementation can be found in our Github repository. [6]

3.2 Iterative approach

Flink programs implement iterative algorithms by defining a step function and embedding it into an iteration operator to execute iterations in a massively parallel fashion. Here we use the **Delta Iterate** operator to implement the PrefixSpan algorithm. The Delta Iterate model is explained below.



1. Iteration Input: The initial workset and solution set are read from data sources or previous operators, as input to the first iteration.

2. Step Function: The step function will be executed in each iteration.

3. Next Workset/Update Solution Set: The next workset drives the iterative computation and will be fed back into the next iteration. Furthermore, the solution set will be updated and implicitly forwarded.

4. Iteration Result: After the last iteration, the solution set is written to a data sink or used as input to following operators. The default termination condition for delta iterations is specified by the empty workset convergence criterion and a maximum number of iterations. The iteration will terminate when a produced next workset is empty or when the maximum number of iterations is reached. It is also possible to specify a custom aggregator and convergence criterion.

Implementation

After we got our sequence database with ordered itemsets from the InputDecoder we flatMap the database to generate the initial workset for DeltaIteration. The InitialWorkSetExpander creates an entry for the first occurrence of each item in a sequence and stores it as Tuple5 ⟨prefix, postfix, 1, joinPrefix, joinParentPrefix⟩.
Prefix and postfix are stored as Integer arrays, where the prefix is the single item and the postfix is the rest of the sequence after the prefix. JoinPrefix and joinParentPrefix are String representations of the prefix and the parent prefix, and are used to achieve better joining performances.
The solution set is initialized as empty dataset of the format Tuple4 ⟨prefix, count, joinPrefix, joinParentPrefix⟩.

With the initial workset and solution set the frequent pattern mining DeltaIteration is started and proceeded until the empty workset convergence criterion is met. The step function itself is divided into three steps.

Step 1: Create the solution set. In this step the workset is grouped by its joinPrefix and the support of each prefix is summed up. Afterwards the infrequent prefixes, which don't meet the threshold, are filtered out. The remaining frequent patterns define the solution set of this iteration.

Step 2: Create the purify set. A DataSet ⟨joinParentPrefix, baseItemSet⟩ which holds the frequent length 1 items of each parent prefix is created. Therefore the solution set is grouped by its joinParentPrefix and the last prefix items are extracted and stored as baseItemSet.

Step 3: Create the new workset. To create the workset for the next iteration three steps are executed.
First, the infrequent prefixes are filtered out by joining by the solution set's and workset's joinPrefix.
Second, the infrequent base items of each postfix are filtered out by joining by the purify set's and workset's joinParentPrefix and by filtering those postfix items which aren't in the baseItemSet.
This significantly reduces the amount of further projections, which are performed in the third step. Here each remaining Tuple gets flatMapped similar to the initial workset projection. For each first occurrence of each item in the postfix a new Tuple5 ⟨prefix+item, postfix, 1, joinPrefix+item, joinPrefix⟩ is created.

After the DeltaIteration has finished the found frequent patterns are collected, ordered by its prefixes and written to a result file.

The source code of this implementation can also be found in our Github repository. [7]

4 Experimental evaluation

There are two factors of our algorithms we wanted to evaluate, correctness and performance. The correctness criterion was met by manually comparing the results of the three sample databases of the DummyInputDecoder with our result files.
To evaluate how performant our algorithm worked we compared it to Apache Spark's reference implementation, with and without pseudoprojection, [8] and a local implementation of the PrefixSpan algorithm in pure Java. [9]
To be comparable we transformed the BigBench datasets of scale factor 1 and 100 into our plain data format, which can be used by all four implementations without further pre-processing. The 88.5MB file of scale factor 1 was transformed to a plain file of 3.9 MB and the 14.7GB file of scale factor 100 was transformed to a plain file of 689MB. The execution times of each implementation can be found in the table below.

	sf 1	sf 100
Pure Java	3.3 sec	1.1 h
Flink Recursive	2385 sec	—
Flink Iterative	73 sec	Exception
Spark with pseudoprojection	36 sec	Exception
Spark without pseudoprojection	81 sec	> 9 h

As the recursive approach lasted disproportionate long at scale factor 1, we resigned to execute it at scale factor 100. We terminated the Spark execution without pseudoprojection after 9 hours for the same reason. The exceptions thrown at scale factor 100 were both runtime exceptions caused by exceeding the hard drive memory.
The Flink and Spark implementations were executed at the "IBM Cluster" of TU-Berlin. 48 executor cores were assigned to each task, which was executed on 8 processing nodes in parallel. At least 10GB of dedicated RAM was provided by each processing node to execute the implementations.
The pure Java implementation was executed on a single Intel Core i5 system with 4GB dedicated RAM and one dedicated core for the execution.

5 Conclusion

Both the Flink and Spark implementations aren't yet able to process large amounts of sequences. They both state runtime exceptions due to exceeding the hard drive memory, even if enough physical memory is provided. The most propable reason we found were serialization issues caused by the large worksets created during the flatMap phases.

As a result we weren't able to compare Spark's and Flink's performance sufficiently and could only deduct from the execution times of sacle factor 1. They state out that pseudoprojection is much more efficient than MapReduce techniques if the datasets fit into main memory. Also the combination of MapReduce for larger sequences and pseudoprojection for those which fit into RAM is more efficient than just using MapReduce techniques.
If we compare just the MapReduce capabilities of Spark's implementation and Flink's iterative one, Flink was around 10% faster than Spark. Our recursive implementation was the least efficient, because Flink needed to read the whole sequence database again from the data source in each recursion call.

Missing Flink features

To implement a more efficient algorithm we missed some key functionalities, which aren't yet provided by Flink.

To improve the performance of our recursive approach, a feature to explicitly write the whole sequence database into RAM, like Spark's ".persist(StorageLevel.MEMORY_AND_DISK)", would be helpful.

To follow a better divide and conquer approach during the flatMap phase, a feature to iterate over each single item in a dataset and perform MapReduce functions on this single item would be better for our algorithm. Spark has an ".forearch()" function for this purpose and uses it in its PrefixSpan implementation.

Spark also offers the opportunity to define unique item ids for better joining performance. We first used the prefix integer array as join key, but it turned out to be inefficient. Therefore we needed to introduce a String representation of the prefix to achieve a better joining performance with the drawback of redundant data.

The last problem we faced was to append information to a dataset outside of our delta iteration and to use it as a data sink afterwards. It's currently not supported by Flink. Therefore we couldn't finish our implementation with local pseudoprojection [10] in an efficient way.[1]

[1]The former commit has a work-around, which is executable but slower.

References

[1] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M. Hsu, Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach

[2] Apache Flink - Homepage, `https://flink.apache.org/`, last accessed 10.02.2016

[3] Apache Spark - Homepage, `https://spark.apache.org/`, last accessed 10.02.2016

[4] Big Data Benchmark for BigBench, `https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench`, last accessed 07.02.2016

[5] Download site of IBM's quest data generator, `http://forum.ai-directory.com/read.php?5,33`, last accessed 07.02.2016

[6] GitHub Repository - Recursive Implementation, `https://github.com/sneJ-/IMPRO-3.WS15/blob/assignment-3/prefixSpan/src/main/java/prefixSpan/PrefixSpan.java`, last accessed 10.02.2016

[7] GitHub Repository - Iterative Implementation, `https://github.com/sneJ-/IMPRO-3.WS15/blob/assignment-3/prefixSpan/src/main/java/prefixSpan/PrefixSpanDelta7.java`, last accessed 10.02.2016

[8] Reference implementation of PrefixSpan in Apache Spark, `https://spark.apache.org/docs/1.6.0/mllib-frequent-pattern-mining.html#prefixspan`, last accessed 08.02.2016

[9] GitHub Repository - Pure Java Implementation, `https://github.com/sneJ-/IMPRO-3.WS15/blob/assignment-3/prefixSpan/src/main/java/prefixSpanPrototype/PrefixSpanPrototypePlain.java`, last accessed 08.02.2016

[10] GitHub Repository - Iterative Pseudoprojection Implementation, `https://github.com/sneJ-/IMPRO-3.WS15/blob/assignment-3/prefixSpan/src/main/java/prefixSpan/PrefixSpanDeltaLocal.java`, last accessed 14.02.2016