Software Design Description

for

DAKA

Version 1.3

April 20, 2014

Prepared by:

Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla,
ByeongGil Jeon and Jirakit Songprasit

Stakeholder: iPipeline

Advisors: Mr. William M. Mongan, Dr. Jeffrey Popyack

Drexel University

Revision History

Name	Date	Reason for changes	Version
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	January 13, 2014	Initial version	1.0
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	February 13, 2014	Included architecture and context diagrams. Description of the algorithms used.	1.1
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	February 18, 2014	Included UML class diagrams for algorithms. Final annotations. Completed glossary	1.2
Sergey Matskevich, Christopher Harvey, Ernesto Cejas Padilla, ByeongGil Jeon and Jirakit Songprasit	April 20, 2014	Added full FP-growth UML and descriptions, updated algorithm description for map-reduce framework. Updated Bayesian Classification UML and descriptions, updated glossary. Updated UML for the interface, added interface descriptions.	1.3

Table of Contents

1 Introduction

1.1 Purpose

This document specifies the entire software architecture for DAKA. The design decisions outlined below comply with the software constraints and functionality requirement outlined in the DAKA Software Requirements Specification document.

iPipeline leads its industry in providing the next-generation suite of sales distribution software to the insurance and financial services markets through its on-demand service. iPipeline is processing about 100 TB of data every week. The processing time it takes to produce results is too long for the business and a solution is required to improve processing time, as well as to account for future increased amounts of data to be processed.

DAKA utilizes horizontally scalable distributed system Hadoop and implement algorithms for it in order to achieve required performance. For this DAKA is deployed on a cluster where number of nodes can grow as needed to meet new processing requirements.

1.2 Scope

This document describes the software architecture for the initial release of DAKA, version 1.0. The intended audience of this document exclusively includes the designers, the developers, and the testers of the DAKA software system.

1.3 Glossary

Attribute

An attribute is a data field representing a characteristic or feature of a data object.

Classifier

A classifier is a model, describing data classes. This model is constructed to predict categorical (discrete) class labels. We want to find out what type of life insurance a customer is likely to by, so product type is used as a classification label.

Class Conditional Independent Assumption

Two events, R and B, are conditionally independent given a third event, Y, precisely if the occurrence or non-occurrence of R and the occurrence or non-occurrence of B are independent events in their conditional probability distribution given Y.

Closed Itemset

An itemset with no proper superset that has the same support count.

Data Classification

A two-step process, consisting of a learning step (where a classification model is constructed) and a classification step (where the model is used to predict class labels for given data).

Database System or Database Management System (DBMS)

It is a collection of interrelated data, known as a database, and a set of software programs to manage and access the data. The software programs provide mechanisms for defining database structure and data storage.

Feature

Any value for one of the columns of the data is considered a feature.

Frequent Itemset

An itemset that occurs in the data with specified minimum support.

Gaussian Distribution

A very commonly occurring continuous probability distribution—a function which tells the probability of an observation in some context to fall between any two real numbers.

Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Hadoop scales up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high availability, the library detects and handles failures at the application layer, so delivering a highly available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop Distributed File System (HDFS)

A distributed file system designed to run on commodity hardware.

Itemset

A set of items in a transaction.

Map

A Map function is a higher-order function that applies to each element of a list, returning a list of results.

Maximal Itemset

Itemset that is not a subset of any other itemset.

Mean

The most common and effective numeric measure of the "center" of a set of data is the (arithmetic) mean. Let $x_1, x_2 \ldots x_N$ be a set of N values or observations, such as for some numeric attribute X, like salary. The mean of this set of values is

$$\text{Mean} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x1+x2+\cdots+xN}{N}.$$

Mining or Data Mining

It is the computational process of discovering patterns in large data sets involving methods at the intersection of artificial intelligence, statistics and database systems.

Reduce

A Reduce function is a higher-order function that recombines elements of a list, returning a single value.

Sequence File

Internal Hadoop file type. It's a flat file consisting of binary key/value pairs.

Standard Deviation

In statistics and probability theory, the standard deviation (SD) (represented by the Greek letter sigma, σ) shows how much variation or dispersion from the average exists. A low standard deviation indicates that the data points tend to be very close to the mean (also called expected value). A high standard deviation indicates that the data points are spread out over a large range of values.

Statistical Classifiers

In machine learning and statistics, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.

Training Data

A set of data objects for which the class labels are known.

Training Tuples

The individual tuples making up the training set are referred to as training tuples.

Transaction

A row of input to DAKA or a tuple in database.

Tuple

A tuple is an ordered list of elements. In set theory, an (ordered) *n*-tuple is a sequence (or ordered list) of *n* elements, where *n* is a non-negative integer.

Venn Diagram

A diagram that shows all possible logical relations between a finite collection of sets. It is used to teach elementary set theory, as well as illustrate simple set relationships in probability, logic, statistics, linguistics, and computer science.

1.4 Context Diagram

The context diagram in Figure 1: Context Diagram shows how the main components of the DAKA system process interact. An external system produces data. DAKA's IO components load that data into the Hadoop framework. DAKA's Compute components process the data. Then DAKA's IO components export that data which an external system consumes.

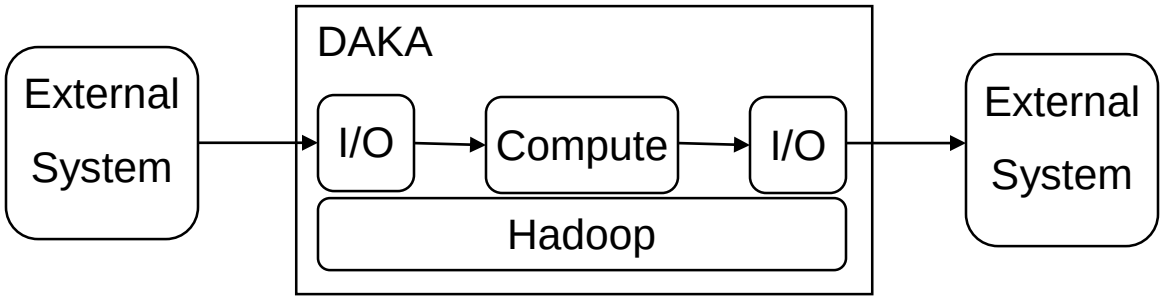


Figure 1: Context Diagram

2 Architecture

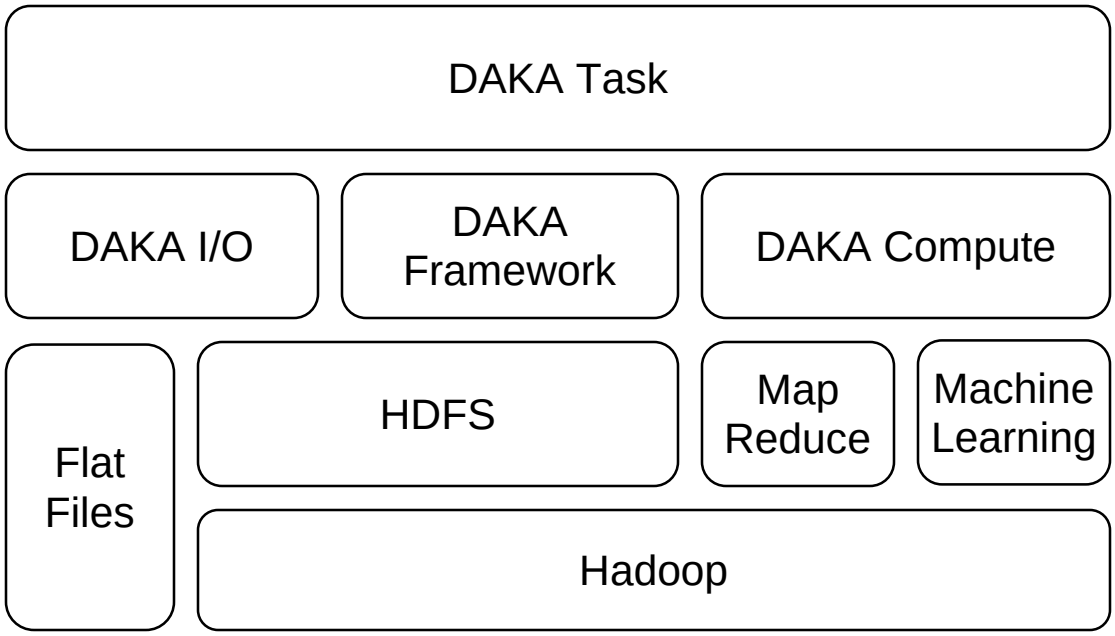


Figure 2: Architecture Diagram

2.1 Overview

DAKA's architecture is component based to maximize the separation of concerns for each module. Figure 2: Architecture Diagram shows the architecture diagram which lays out the components and how they interact.

2.2 Technologies Used

DAKA uses HDFS to store large amounts of data, distributed across several machines. DAKA uses Apache Hadoop for sorting and analyzing the data by implementing the MapReduce computation model.

2.2.1 HDFS

HDFS is a distributed and scalable file system written for the Hadoop framework. Figure 3: HDFS Architecture shows the architecture of an HDFS instance. A single instance of HDFS consists of a single namenode and multiple datanodes. The namenode tracks the metadata of the files being stored in the file system. This includes the locations of each piece of data. Each of the datanodes stores a portion of the data.

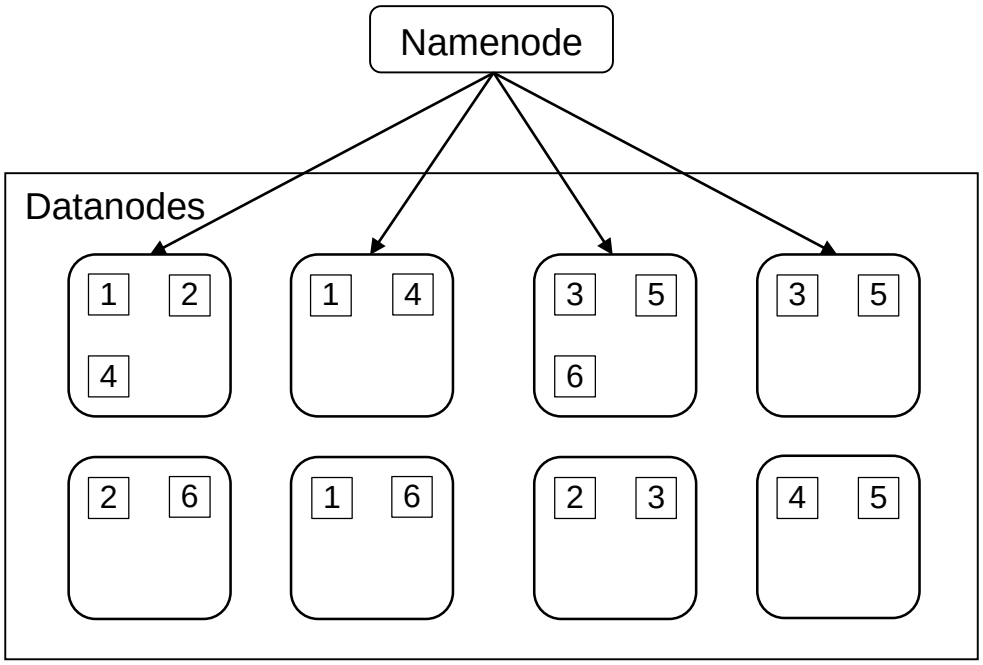


Figure 3: HDFS Architecture

HDFS provides reliability by replicating data across the datanodes. By default, there are three replications of each piece of data. HDFS also provides scalability by allowing for the addition of new datanodes. Each additional datanode adds additional storage space for the HDFS cluster.

2.2.2 Hadoop

Hadoop is a software framework that allows for processing large-scale data. It uses HDFS to store input data across a cluster of machines. Hadoop then manages the execution of programs written using the MapReduce paradigm. Finally, the results of the execution remain stored in HDFS. Figure 4: Hadoop execution flow shows the flow of Hadoop's execution.

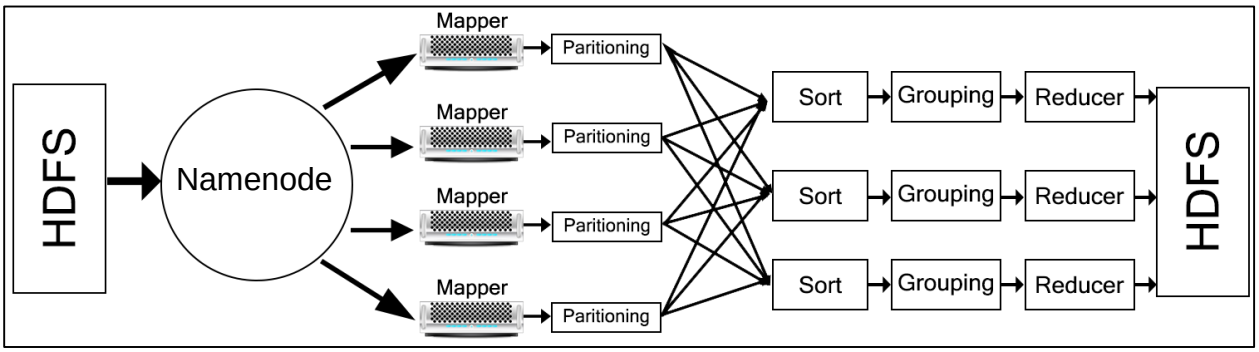


Figure 4: Hadoop execution flow

Because Hadoop uses HDFS, all data to be processed exists across multiple datanodes. Hadoop handles the execution of processing by distributing the execution across the nodes where the data exists. By moving the execution to the data, Hadoop reduces traffic between nodes by eliminating the need to transfer input data. In addition, data replication allows Hadoop to decide which copy of data to execute on to reduce load on individual datanodes.

The MapReduce paradigm allows for parallel execution of data. Hadoop manages this parallel execution by first executing the Map step on the input data where it exists in HDFS. Hadoop then transfers the intermediate results from the Map step to datanodes for the Reduce step. It then stores the result of the Reduce step in HDFS by replicating the data across nodes. By managing the execution of a program written with the MapReduce paradigm, Hadoop simplifies the work necessary to take advantage of distributed processing.

2.3 Framework Components

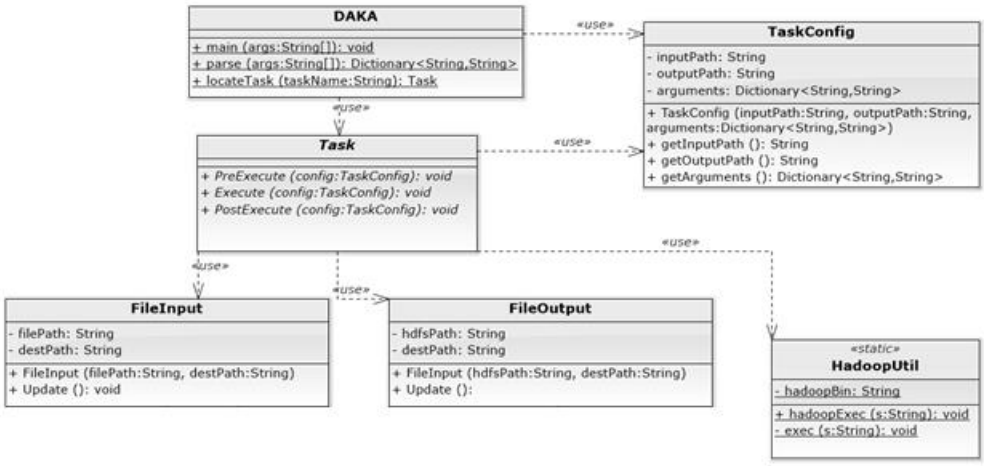


Figure 5: Overview of framework components

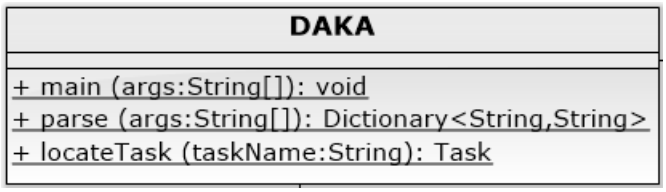
2.3.1 Overview

The DAKA framework consists of components that allow for the execution of DAKA software. This includes the interface that the user interacts with, the task model used to designate specific scenarios, as well as general utilities used through DAKA.

2.3.2 User Interface

2.3.2.1 DAKA

The DAKA component serves as the main entry point of the framework.

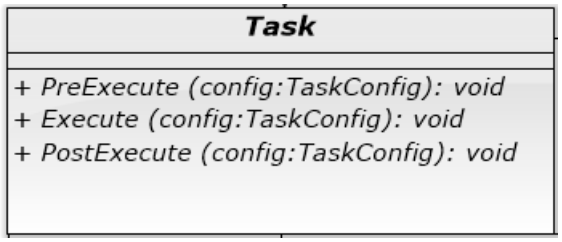


- main – the entry point of the DAKA framework
- parse – parses the command line arguments to form a dictionary of arguments
- locateTask – locates and instantiates a task with the given name from a directory of task jars

2.3.3 Task Model

2.3.3.1 Task

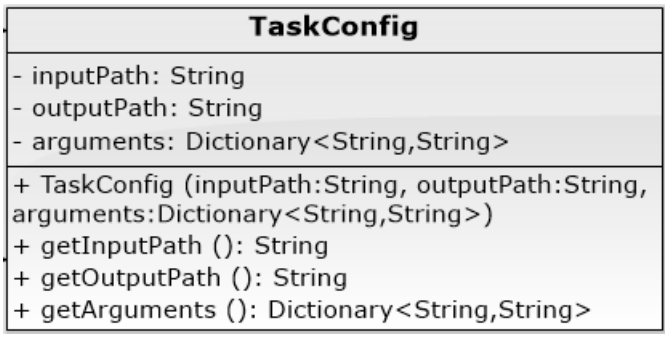
The Task model is an abstraction that is implemented by user-defined tasks. The DAKA framework calls the three steps in the Task definition. Each step has an intended use but it is up to the user to define the purpose of each step.



- PreExecute – first method called intended for preparing input data
- Execute – second method called intended for running the bulk of the data processing
- PostExecute – third method called intended for handling output data

2.3.3.2 Task Configuration

The Task Configuration contains the collection of arguments passed to a Task at runtime. These arguments include the location of input data and the destination of output data. The collection also includes any task specific arguments.

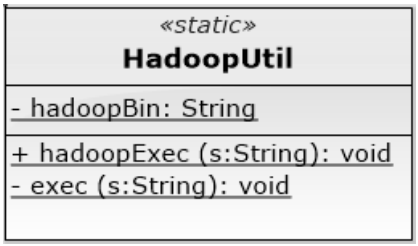


- inputPath – the path entered at the command line for input data
- outputPath – the path entered at the command line for output data
- arguments – all arguments passed to DAKA at the command line
- TaskConfig – constructor which accepts values for the inputPath, outputPath, and arguments fields
- getInputPath – accessor for inputPath field
- getOutputPath – accessor for outputPath field
- getArguments = accessor for arguments field

2.3.4 General Utilities

2.3.4.1 Hadoop Utility

The Hadoop Utility component handles interactions with the Hadoop framework.



- hadoopBin – the path to the Hadoop binary
- hadoopExec – executes Hadoop commands using the Hadoop binary
- exec – executes command line commands

2.4 Input and Output Components

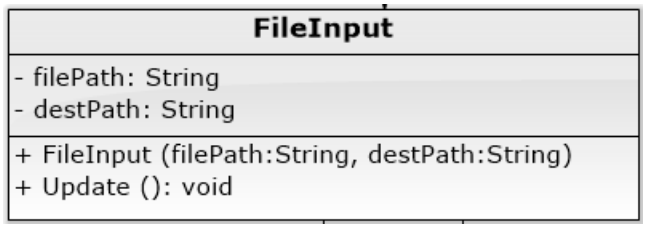
2.4.1 Overview

DAKA's input and output components help manage the data used as input for a DAKA task and the data resulting from the execution of the task.

2.4.2 Input

2.4.2.1 File Input

The File Input component facilitates using local files as inputs to a DAKA task. The component transfers a local file into HDFS when necessary.

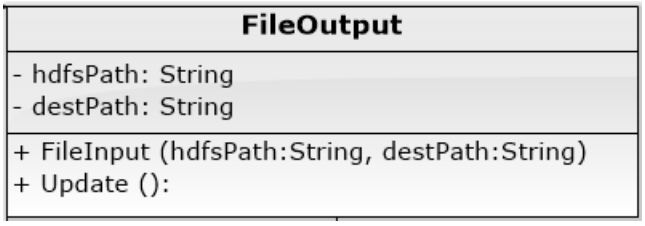


- filePath – the path in the local filesystem where the input exists
- destPath – the path in HDFS where the input is copied
- FileInput – constructor which accepts values for the filePath and destPath fields
- Update – if local file is newer than destination file, replaces destination file with local file

2.4.3 Output

2.4.3.1 File Output

The File Output component facilitates storing the results of a DAKA task into a local file. The component transfers the results from HDFS into a local file.



- hdfsPath – the path in HDFS where the output exists
- destPath – the path in the local filesystem where the output is copied
- FileOutput – constructor which accepts values for the hdfsPath and destPath fields
- Update – if HDFS file is newer than destination file, replaces destination file with HDFS file

2.5 Compute Components

2.5.1 Overview

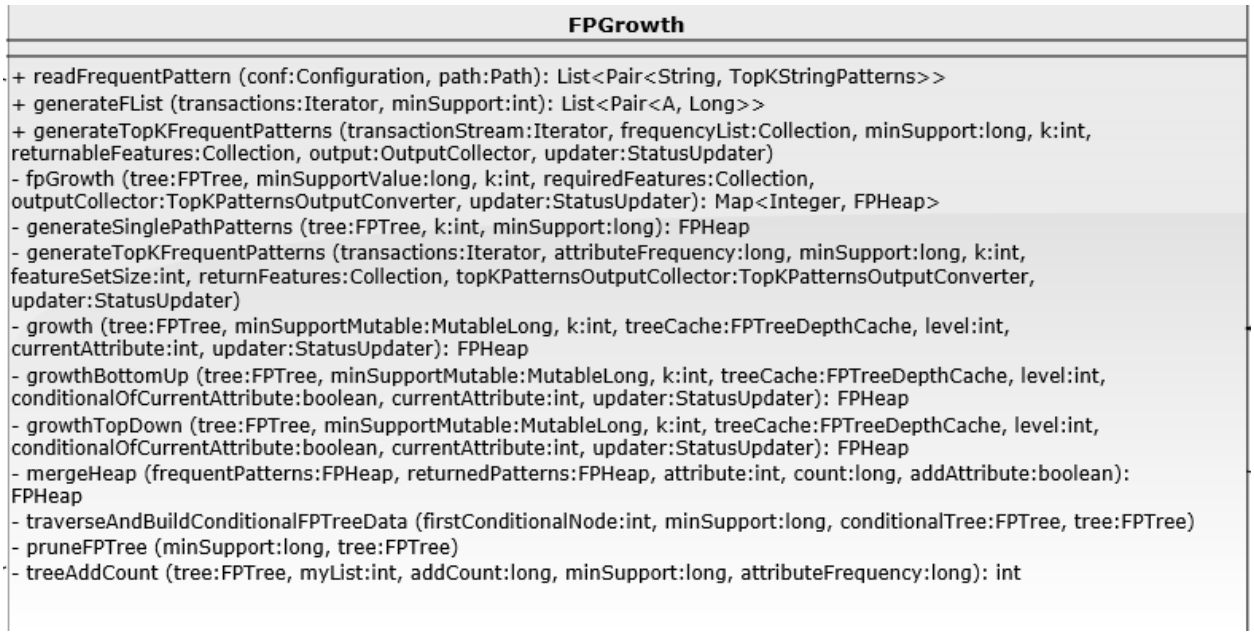
DAKA includes implementations of algorithms written to take advantage of the Hadoop framework. These implementations allow Hadoop to distribute execution by using the

MapReduce paradigm when possible.

2.5.2 FP-growth

2.5.2.1 FP-growth

The FP-growth object performs the FP-growth algorithm.

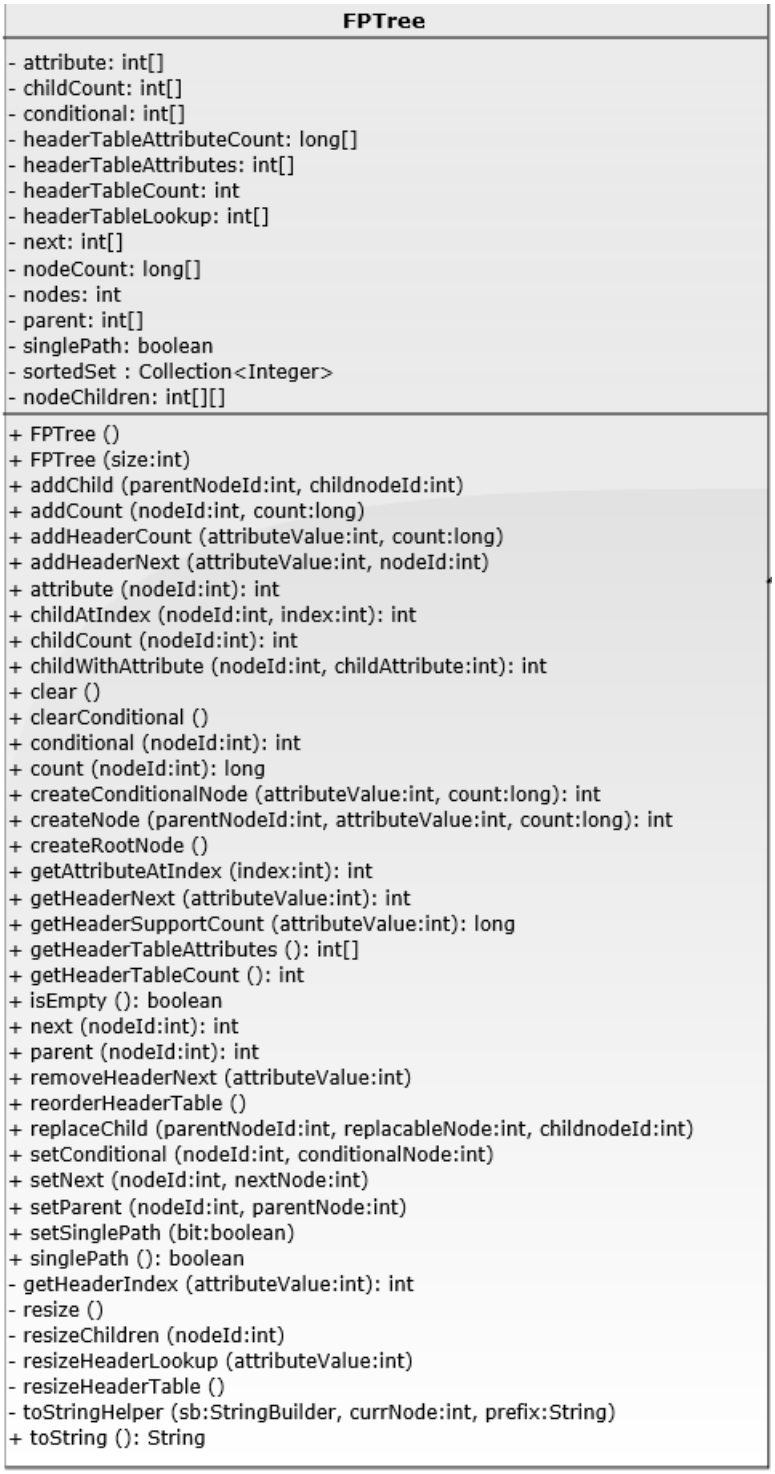


- readFrequentPattern – reads the stored frequent patterns from the given location
- generateFList – generates the feature frequency list from the given transaction whose frequency is greater than minSupport
- fpGrowth – runs the FP-growth algorithm to generate the FP-tree with minimum support value
- generateSinglePathPatterns – generates a single path pattern with the given minimum support
- generateTopKFrequentPatterns – generates top K frequent patterns for every feature in returnable features given a stream of transactions and the minimum support
- growth - generates patterns for each item. Internally called from fpGrowth
- growthTopDown - generates patterns from conditional tree. Called internally from growth
- growthBottomUp - used by growthTopDown to generates patterns from conditional tree for each attribute.

- mergeHeap - merges sets of patterns during growth
- traverseAndBuildConditionalFPTreeData - builds conditional FPTree
- pruneFPTree - prunes subsets of frequent patterns closed maximal sets
- treeAddCount - create FPTree with node counts, returns total number of children in the tree. Used internally by generateTopKFrequentPatterns

2.5.2.2 FP-tree

An FP-tree is a representation of the tree used to store the attributes analyzed by the FP-growth algorithm.



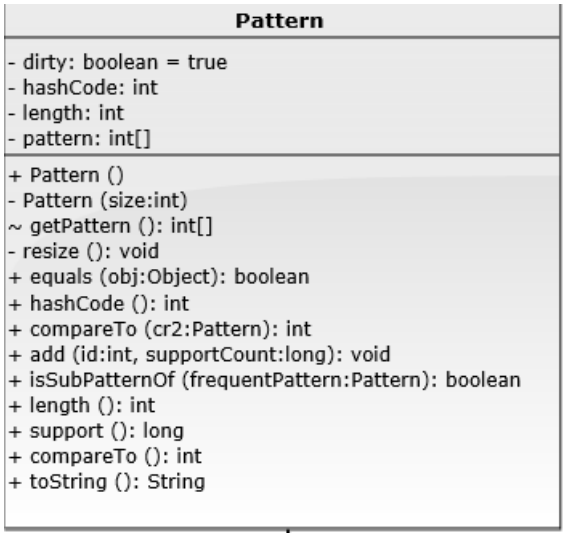
- attribute – a collection of attribute identifiers
- childCount – the count of children of a node
- conditional – a collection of conditions

- headerTableAttributeCount – the count of header table attributes
- headerTableAttributes - list of attributes in the header table
- headerTableCount – the count of header tables
- headerTableLookup – the lookup table for headers
- next – a collections of pointers to the next nodes
- nodeCount – the collection of counts of nodes for each node
- nodes – the total count of all nodes
- parent – a collection of pointers to parent nodes
- singlePath – indicates whether the tree is a single path
- sortedSet - list of attributes in sorted order for faster operations
- nodeChildren - list of conditional attributes for nodes
- FPTree – constructs an empty FP-tree
- FPTree – constructs an FP-tree of given size
- resize – resizes the tree to include more elements
- getHeaderTableAttributes – returns attributes from the header table
- replaceChild – replaces the child of the specified node
- resizeHeaderTable – resizes table used to store headers
- clear - resets the whole tree
- addChild - add a child to a node
- addCount - add count to nodeCount item
- addHeaderCount - add count to headerTableAttributeCount item
- addHeaderNext - sets successor value
- attribute - returns attribute value
- childAtIndex - return child value
- childWithAttribute - finds index of a child for given attribute
- childCount - returns childCount
- clearConditional - clears the list of conditionals
- conditional - returns the conditional for given node
- count - return count for given node
- createConditionalNode - adds conditional child to the tree

- createNode - add new node to header nodes
- createRootNode - create new root node for current tree
- getAttributeAtIndex - returns an attribute value for given index
- getHeaderNext - returns successor for given attribute
- getHeaderSupportCount - returns support count for given attribute
- getHeaderTableCount - returns number of items in the header table
- isEmpty - checks if the tree is empty
- next - gets next node for given id
- parent - return parent for given node
- removeHeaderNext - removes successor for given attribute
- reorderHeaderTable - sorts header table
- setConditional - assign value to a conditional for a given node
- setNext - sets new next pointer
- setParent - sets new parent for a given node
- setSinglePath - assigns value to singlePath
- singlePath - returns singlePath
- getHeaderIndex - returns index of a given attribute.
- resizeChildren - increase size of child list for given node
- resizeHeaderLookup - increase size of headerTableLookup
- resizeHeaderTable - increase size of the header table
- toStringHelper - used by toString
- toString - produces string representation of the tree

2.5.2.3 Pattern

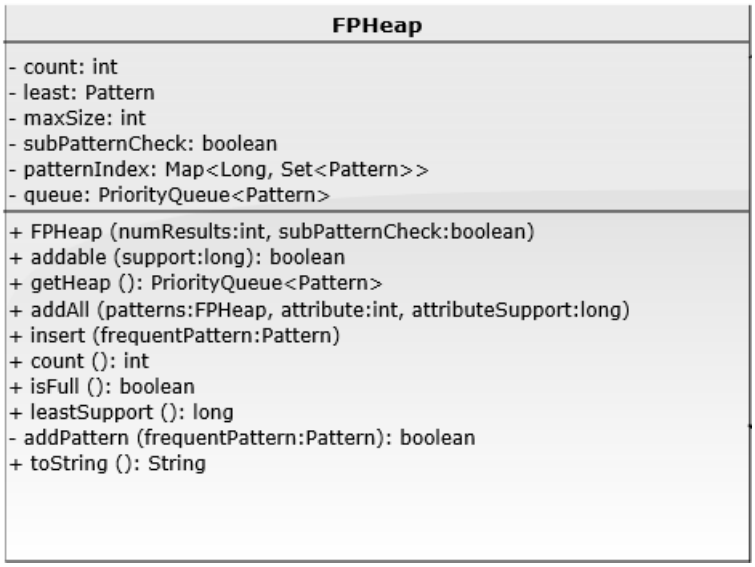
A Pattern is a list of items and their support, which is the number of times the Pattern appears in the data set.



- dirty – a Boolean that indicates the hash has not been computed
- hashCode – the hash code of the Pattern for comparisons
- length – the number of elements in the Pattern
- pattern – the elements of the Pattern
- support – the number of occurrences in the database
- Pattern – constructs an empty Pattern
- Pattern – constructs a Pattern with the given size
- resize – increases capacity if the Pattern
- equals – compares the Pattern to another object for equality
- hashCode – returns the hash code of the Pattern
- isSubPatternOf – checks if current pattern is a sub-pattern of another pattern
- support - returns support for the pattern
- compareTo - compares length and support of a pattern to another pattern
- toString - returns string representation of current pattern
- add - adds new attribute to the pattern
- equals - deep equality check for patterns

2.5.2.4 Frequent Pattern Heap

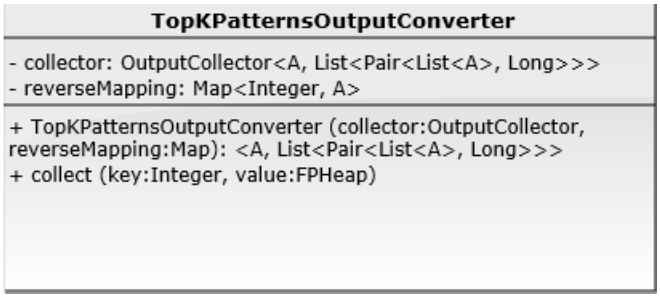
Frequent Pattern Heap is a priority queue that keeps top *k* attributes in a tree set.



- count – stores the number of Patterns stored in heap
- least – stores the Pattern with minimum support
- subPatternCheck - flag to indicate search for maximal closed pattern set
- patternIndex - hash tables of patterns for quick access
- queue - max ordered priority queue of patterns
- FPHeap – constructs the heap with room for the given number of Patterns
- addable – checks if this pattern can be added to the list
- getHeap – returns the priority queue containing the Patterns
- addAll – adds all of the Patterns from a separate FPHeap to the current FPHeap
- insert – inserts a Pattern into the collection
- count – accessor for number of Patterns stored in heap
- isFull – returns a Boolean indicating if the heap is full
- leastSupport – returns minimum support parameter
- addPattern – adds a pattern to the internal Priority Queue, used internally by insert
- toString – converts the Frequent Pattern Max Heap to a string

2.5.2.5 Top K Patterns Output Converter

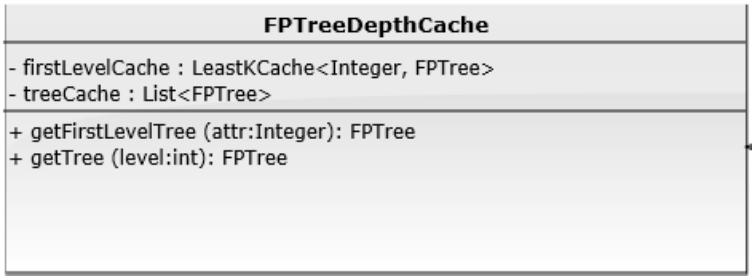
Top K Patterns Output Converter is an output converter that converts the output patterns and collects them in a Frequent Pattern Max Heap.



- collector - Hadoop OutputCollector
- reverseMapping - pattern-support reverse map
- TopKPatternsOutputConvert - constructs the Top K Patterns Output Converter with the given Output Collector and the Map function
- collect - conerts FPHeap to list of pairs patter:support count and outputs it to HDFS

2.5.2.6 FP-tree Depth Cache

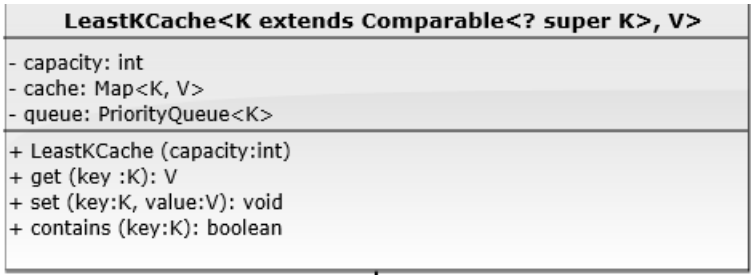
The FP-tree Depth Cache caches a large FP-tree for each level of the recursive algorithm to reduce allocation overhead.



- treeCache – a list of conditional FP-trees for recursive mining
- firstLevelCache - forest of frequent pattern trees
- getFirstLevelTree – creates and returns conditional FP-tree for mining
- getTree - returns FP-tree of appropriate level

2.5.2.7 Least K Cache

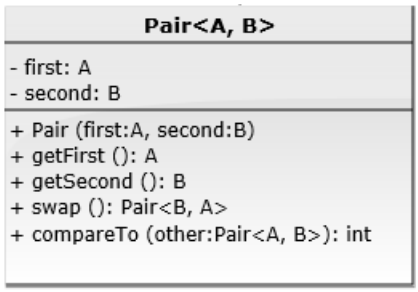
The object represents forest of frequent pattern trees for each recursion level



- capacity - maximum capacity of the cache
- cache - hashmap of pattern trees
- queue - max ordered priority queue of patterns

2.5.2.8 Pair

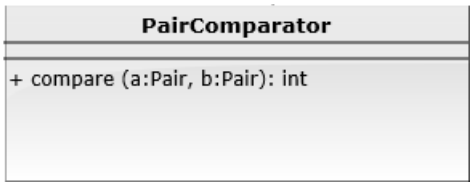
Represents a 2-tuple of objects



- first - first object
- second - second object
- Pair - creates new pair
- getFirst - returns first object
- getSecond - returns second object
- swap - swaps first and second objects
- compareTo - Defines an ordering on pairs that sorts by first value's natural ordering, ascending, and then by second value's natural ordering.

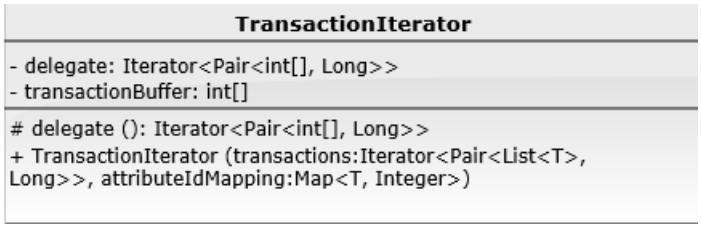
2.5.2.9 Pair Comparator

Defines an ordering on Pairs whose second element is a count. The ordering places those with high count first (that is, descending), and for those of equal count, orders by the first element in the pair, ascending



- compare - compares 2 pairs

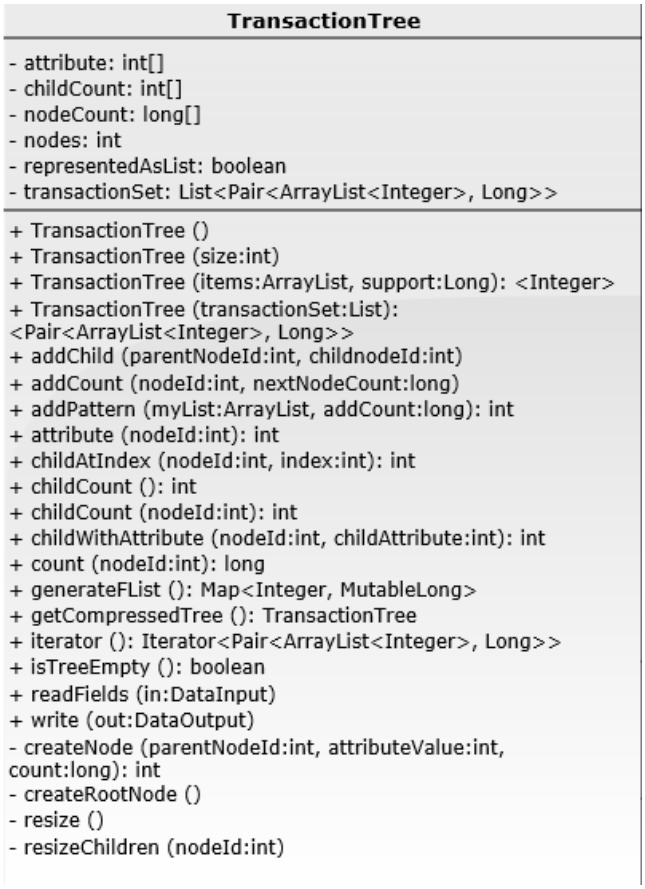
2.5.2.10 Transaction Iterator



- delegate - iterator
- transactionBuffer - list of attributes from current transaction
- TransactionIterator - creates new iterator
- delegate - returns delegate

2.5.2.11 Transaction Tree

Representation of transactions modeled on the lines to FPTree

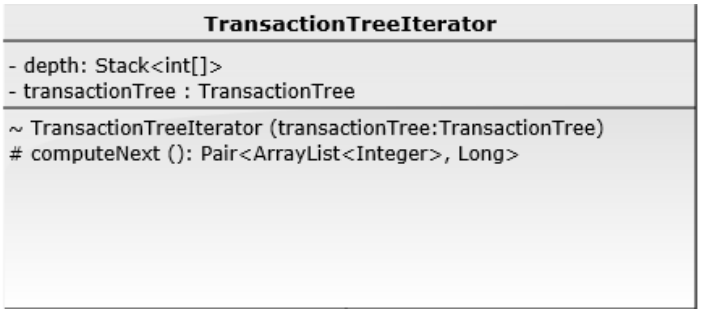


- Representation of transactions modeled on the lines to FPTree
- attribute – a collection of attribute identifiers
- childCount – the count of children of a node
- nodeCount - the collection of counts of nodes for each node
- nodes - total number of nodes
- representedAsList - true if the current transaction have been converted to the list of pairs items:support
- transactionSet list of pairs items:support
- TransactionTree - creates new transaction tree
- TransactionTree - creates new transaction tree of a given size
- TransactionTree - creates new transaction tree from provided list with given support
- TransactionTree - creates new transaction tree from transaction set
- addChild - adds new child to a node

- addCount - add support count for a given node
- addPattern - adds support count for attributes in provided pattern to attributes in the tree
- attribute - get attribute for a given node
- childAtIndex - returns attribute value for a given node at a given index
- childCount - return total number of children
- childCount - return number of children for a given node
- childWithAttribute - return index of a child with provided attribute value
- count - returns support count for given node
- generateFList - generates frequency list for all items in the current tree
- getCompressedTree - creates a tree with duplicates removed and counted
- iterator - transaction iterator
- isTreeEmpty - checks if tree is empty
- readFields - reader of TransactionTree from HDFS
- write - writer of TransactionTree to HDFS
- createNode - adds new node to the tree
- createRootNode - creates root node
- resize - increases capacity of the tree
- resizeChildren - increases children capacity for given node

2.5.2.12 Transaction Tree Iterator

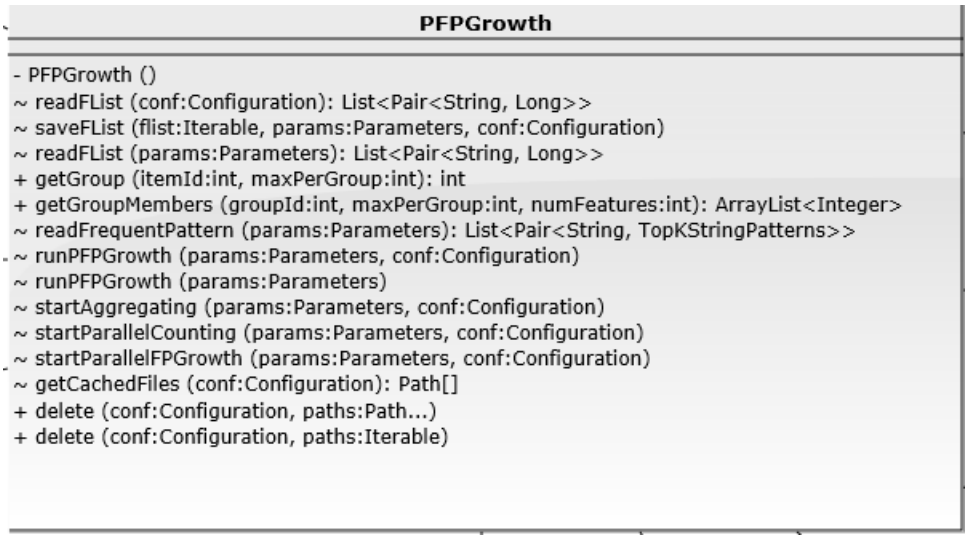
Generates a List of transactions view of Transaction Tree by doing Depth First Traversal on the tree structure



- depth - depth of the current path on the tree
- transactionTree - transaction tree to iterate over

- computeNext - returns next item

2.5.2.13 PFPGrowth

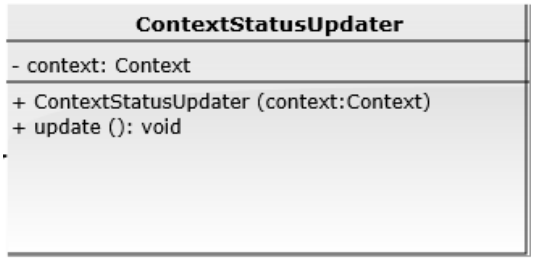


- readFList - Generates the fList from the serialized string representation
- saveFList - Serializes the fList and returns the string representation of the List
- readFList - read the feature frequency List which is built at the end of the Parallel counting job
- getGroup - returns the group item belongs to after split
- getGroupMembers - get item ids in the transaction database for provided group
- readFrequentPattern Read the Frequent Patterns generated from Text
- runPFPGrowth - runs parallel fpgrowth with provided configuration
- runPFPGrowth - runs parallel fpgrowth with default configuration
- startAggregating - Run the aggregation Job to aggregate the different TopK patterns and group each Pattern by the features present in it and thus calculate the final Top K frequent Patterns for each feature
- startParallelCounting - Count the frequencies of various features in parallel using Map/Reduce
- startParallelFPGrowth - Run the Parallel FPGrowth Map/Reduce Job to calculate the Top K features of group dependent shards
- getCachedFiles - Retrieves paths to cached files.
- delete - deletes all files, which paths provided as arguments

- delete - deletes all files from iterable collection provided

2.5.2.14 Context Status Updater

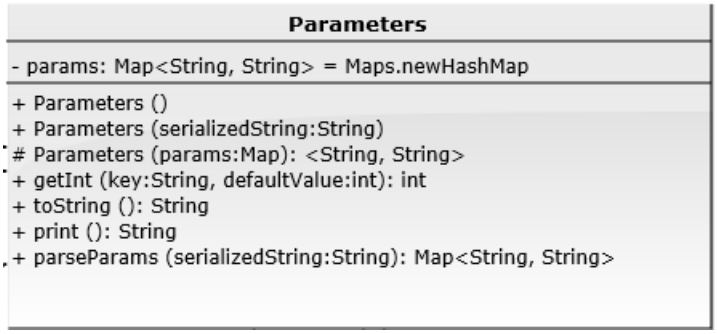
Updates the Context object of a Reducer class



- context - Context
- ContextStatusUpdater - create new instance
- update - update status

2.5.2.15 Parameters

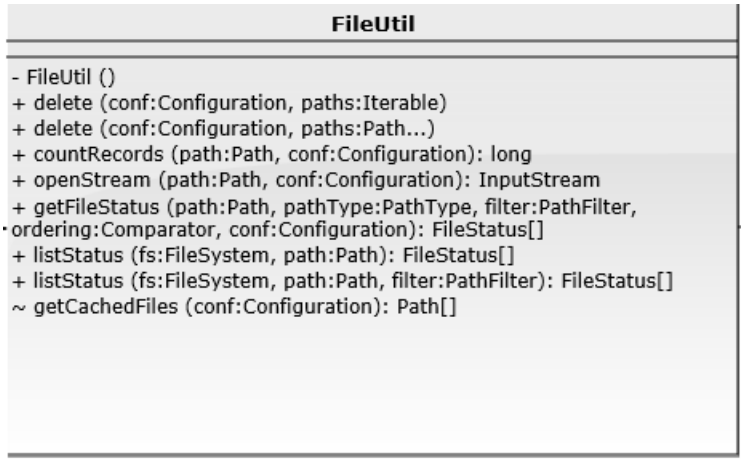
Custom parameters set used internally



- params - parameters map
- Parameters - create new parameters object
- Parameters - create paramters from serialized string
- Parameters - create new object using existing map of parameters
- getInt - returns parameters as an integer
- toString - serializes parameters using hadoop serializer
- print - returns string representation of params map
- parseParams - parses hadoop serialized string of parameters

2.5.2.16 File Util

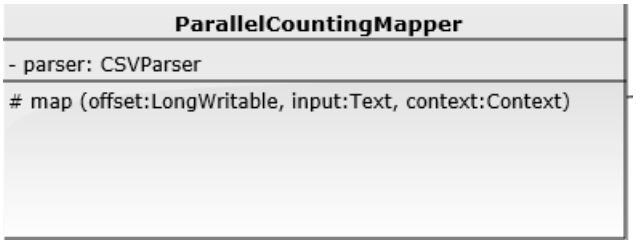
Class for HDFS file operations



- delete - deletes all files, which paths provided as arguments
- delete - deletes all files from iterable collection provided
- countRecords - counts number of entries in Hadoop Sequence File
- openStream - opens input stream for provided file
- getFileStatus - returns file status(es) for specified file or collection of files (uses pattern matching and FS filters)
- getCachedFiles - Retrieves paths to cached files.
- listStatus - used by getFileStatus
- listStatus - used by getFileStatus

2.5.2.17 Parallel Counting Mapper

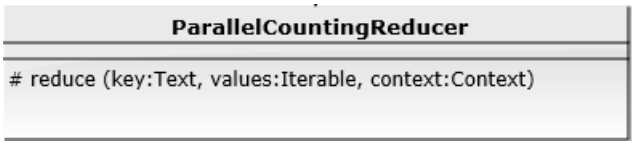
Maps all items in a particular transaction



- parser - csv parser
- map - map function, reads a line and maps all items in the line

2.5.2.18 Parallel Counting Reducer

Sums up the item count and output the item and the count



- reduce - aggregator function for each key in the reducer

2.5.2.19 Aggregator Mapper

Outputs the pattern for each item in the pattern, so that reducer can group them and select the top K frequent patterns

- map - mapper function

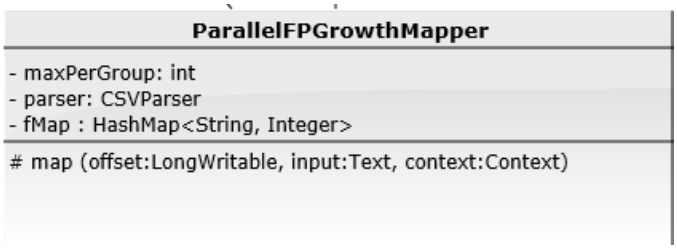
2.5.2.20 Aggregator Reducer

Groups all Frequent Patterns containing an item and outputs the top K patterns containing that particular item

- maxHeapSize - K in top K patterns
- reduce - aggregate patterns
- setup - read K from context

2.5.2.21 Parallel FP Growth Mapper

Maps each transaction to all unique items groups in the transaction. Mapper outputs the group id as key and the transaction as value

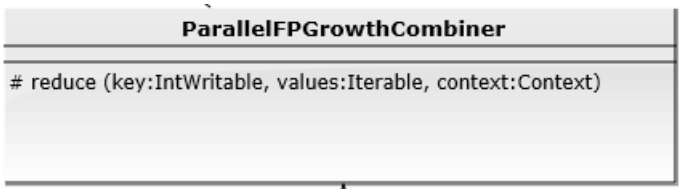


- maxPerGroup - maximum number of transactions per group
- parser - csv parser

- fMap - frequency list of all items
- map - mapper function for transaction

2.5.2.22 Parallel FP Growth Combiner

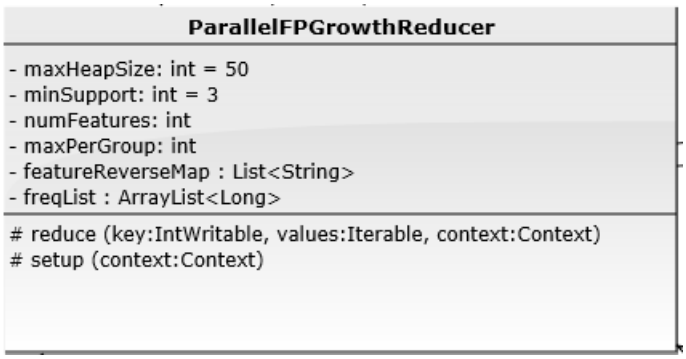
Takes each group of dependent transactions and compacts it in a TransactionTree structure



- reduce - makes one transaction tree from the forest that came from

2.5.2.23 Parallel FP Growth Reducer

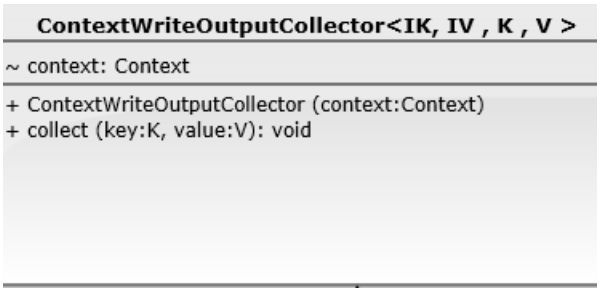
Takes each group of transactions and runs FPGrowth on it and outputs the the Top K frequent Patterns for each group.



- maxHeapSize - K in top K patterns
- minSupport - minimum support
- numFeatures - total number of features in the transaction database
- maxPerGroup - maximum number of transactions per group
- featureReverseMap - reverse map of attributes
- freqList - list of support count for each feature
- reduce - reduce function, runs fpgrowth for incoming transaction tree
- setup - reads feature list and counts, populates class variables from context

2.5.2.24 Context Write Output Collector

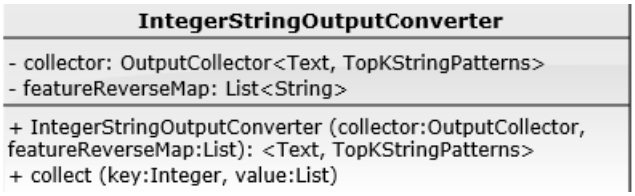
An output collector for Reducer for PFPGrowth which updates the status as well as writes the patterns generated by the algorithm



- context - context
- ContextWriteOutputCollector - creates new output collector
- collect - writes passed parameters to the environment

2.5.2.25 Integer String Output Converter

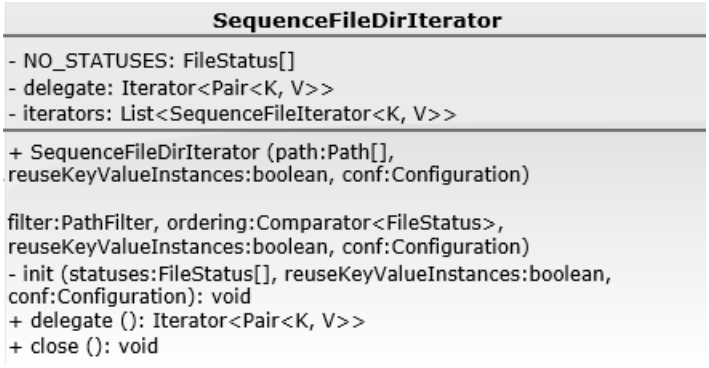
Collects the Patterns with Integer id and Long support and converts them to Pattern of Strings based on a reverse feature lookup map.



- collector - Hadoop OutputCollector
- IntegerStringOutputConverter - create new instance
- collect - write contents to the environment

2.5.2.26 Sequence File Dir Iterator

Iterates over list of Hadoop Sequence Files. The input path may be specified as a directory of files to read, or as a glob pattern. The set of files may be optionally restricted with a PathFilter.

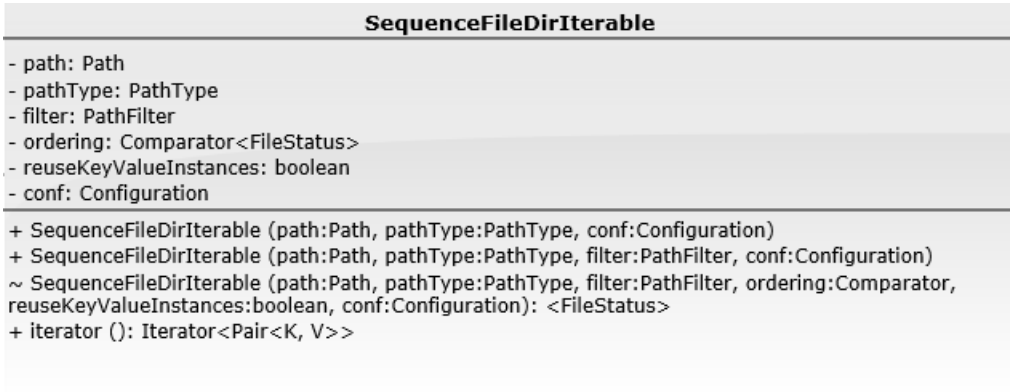


- NO_STATUSES - used for cases when no files matched search criteria

- delegate - iterator for items in the file
- iterators - list of file iterators
- SequenceFileDirIterator - creates new iterator
- init - class initializer, used by constructor
- delegate - returns delegate iterator
- close - close iterator

2.5.2.27 Sequence File Dir Iterable

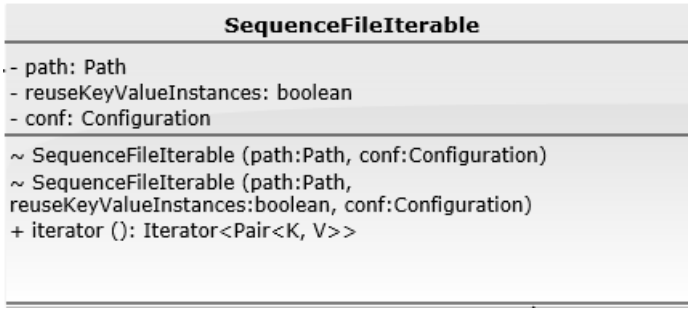
Counterpart to SequenceFileDirIterator



- path - path to directory
- pathType - specifies where this is simple directory path or unix glob pattern
- filter - specifies sequence files to be ignored by the iteration
- ordering - specifies the order in which to iterate over matching sequence files
- reuseKeyValueInstances - if true, reuses instances of the value object instead of creating a new one for each read from the file
- conf - environment configuration
- SequenceFileDirIterable - creates new iterable
- SequenceFileDirIterable - creates new iterable with filter
- SequenceFileDirIterable - full constructor and initializer for the class
- iterator - returns SequenceFileDirIterator for specified path

2.5.2.28 Sequence File Iterator

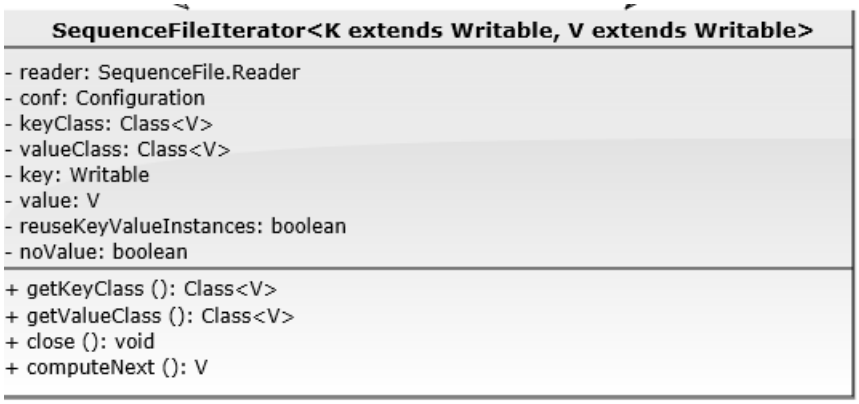
Iterator over a SequenceFile's keys and values, as a Pair



- reader - sequence file reader
- conf - environment configuration
- valueClass - data type of values in the Pair
- keyClass - data type of keys in the Pair
- key - current key
- value - current value
- reuseKeyValueInstances - if true, reuses instances of the value object instead of creating a new one for each read from the file
- noValue - true if current value class is null
- getKeyClass - return keyClass
- getValueClass - return valueClass
- close - close iterator
- computeNext - get next value

2.5.2.29 SequenceFileIterable

Counterpart to SequenceFileIterator



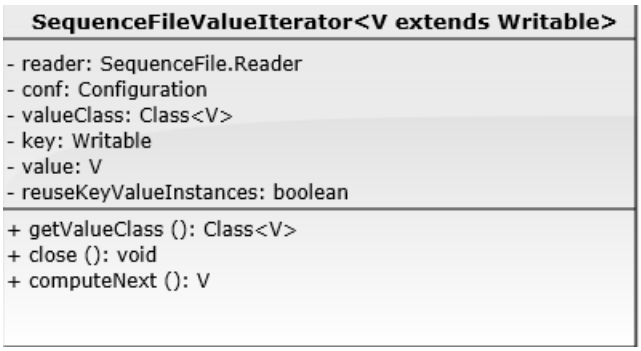
- path - path to directory
- reuseKeyValueInstances - if true, reuses instances of the value object instead of

creating a new one for each read from the file

- conf - environment configuration
- SequenceFileIterable - creates new iterator where reuseKeyValueInstances is false by default
- SequenceFileIterable - creates new iterator with specified reuseKeyValueInstances
- iterator - returns SequenceFileIterator

2.5.2.30 Sequence File Value Iterator

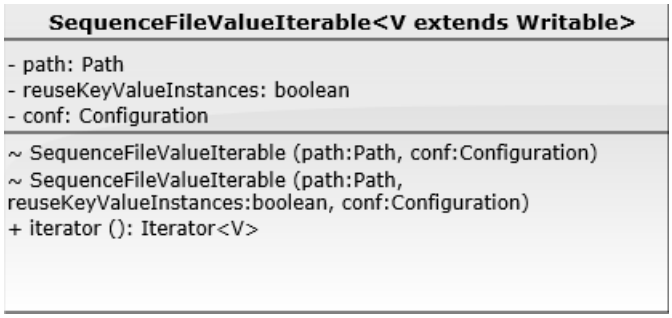
Iterator over a SequenceFile's values only



- reader - sequence file reader
- conf - environment configuration
- valueClass - data type of values in the Pair
- key - current key
- value - current value
- reuseKeyValueInstances - if true, reuses instances of the value object instead of creating a new one for each read from the file
- getValueClass - return valueClass
- close - close iterator
- computeNext - get next value

2.5.2.31 Sequence File Value Iterable

Counterpart to Sequence File Value Iterator

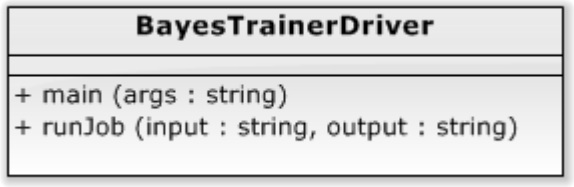


- path - path to directory
- reuseKeyValueInstances - if true, reuses instances of the value object instead of creating a new one for each read from the file
- conf - environment configuration
- SequenceFileValueIterable - creates new iterator where reuseKeyValueInstances is false by default
- SequenceFileValueIterable - creates new iterator with specified reuseKeyValueInstances
- iterator - returns SequenceFileValueIterator

2.5.3 Naïve Bayesian Classification

2.5.3.1 Bayes Trainer Driver

Bayes Trainer Driver is the class that runs a map reduce job to train the Bayes algorithm. It coordinates the Bayes Mapper and the Bayes reducer classes to train the Bayes algorithm.

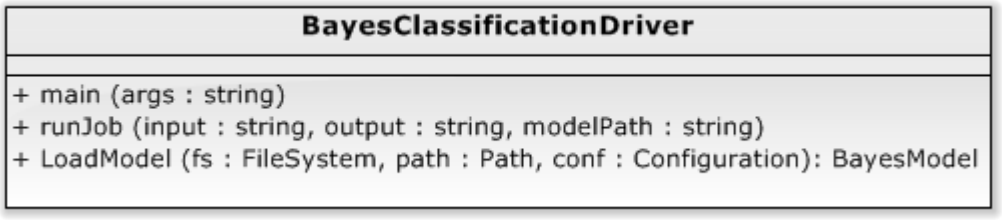


- main – This function receives the input path for the necessary files needed to train the Bayes algorithm. Also receives the output path where the Hadoop Sequence File is stored. It then calls the runJob function.
- runJob – It runs a map reduce job using Bayes Mapper class and Bayes Reducer class to create a Sequence File that is used later to load and create the Bayes Model.

2.5.3.2 Bayes Classification Driver

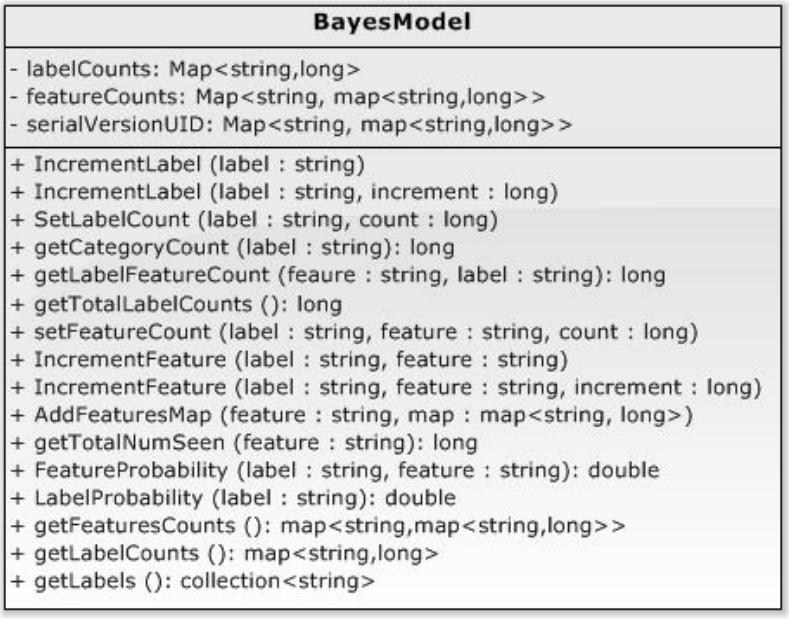
The Bayes Classification Driver runs a map reduce job that calculates the correlation of

the input data to the products.



- main - This function receives the path of the input files to classify in different products. Also receives the output path where the results are stored and the location where the model path is stored. It calls the runJob function with these paths.
- runJob – It runs the map reduce job using the BayesClassificationMapper class and the BayesModel class to calculate the correlation of the input data and all the products.
- LoadModel – It loads the BayesModel class from Hadoop file system.2.5.3.2 Naïve Bayes Model Naïve Bayes Model holds special data structures that contain the information learned in the training step of the Bayes algorithm. The information stored in this class is used to calculate the correlation between new data and existing information gotten in the training part.

2.5.3.3 Bayes Model



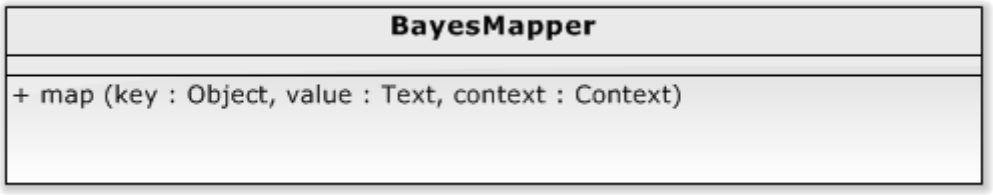
- labelCounts – Is a hashmap that holds the count of labels found in the training.

- featuresCounts – Is a hashmap of hashmap where each feature value found in the training is mapped to the labels it belonged and the count of how many times this happened in the training.
- serialVersionUID – This is used by the system when the class is serialized.

- IncrementLabel – Is used to increase the amount of times a label is used. One is to be increase the label by one and the other is to increase the count of label by the variable increment.
- SetLabelCount – Set the count for a certain label.
- getCategoryCount – Get the count for certain label.
- getLabelFeatureCount – Get the count for a given feature found in certain label.
- GetTotalLabelCounts– Get the count for all the labels.
- setFeatureCounts – Set the count for a given feature.
- IncrementFeature – Increment the count of a feature by one or a given amount.
- numLabels – the amount of properties for the model
- createScoringVector – stores the intermediate probabilities for each feature
- AddfeaturesMap – Adds a Map that holds counts of labels to the data structure featuresCount.
- getTotalNumSeen– Gets the count for a given feature.
- FeatureProbability – Gives the correlation of a feature for a given label.
- LabelProbability – Gives the probability of a random feature to belong to a given label.
- getFeatureCounts – Gets the Map datastructure that holds the counts a labels for each feature.
- getLabelCounts – Get the map data structure that holds the counts for each label.
- getLabels – get all the labels stored in the model.

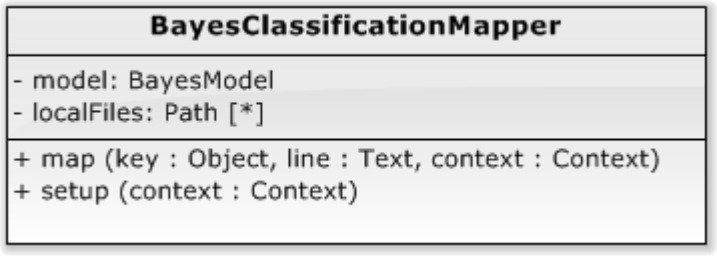
2.5.3.4 Bayes Mapper

The Bayes Mapper class is the mapper of the map-reduce job used in the training part of the algorithm.



- map - Parses the input string into CSV format. For each of the features in the csv line, passes to the reducer tuples of the form <string, string> where the first value is the feature and the second value is the product category the input lines belongs to.

2.5.3.5 Bayes Classification Mapper

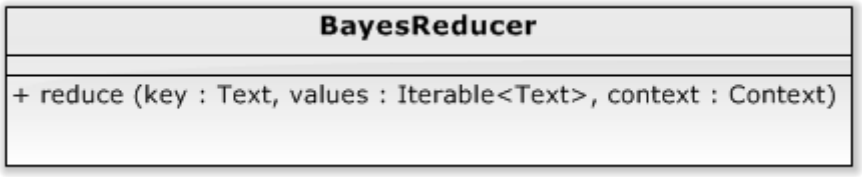


- model – This is the model of the Bayes algorithm used to classify. This variable is initialized in the setup function. The model loaded is the Bayes model that was created in the Bayes Classification Driver class previous to launching the map-reduce job and distributed to the mappers in the hadoop DistributedCache class.
- localFiles – This array object holds the path to load the Bayes model object. The path is found in the hadoop DistributedCache class and the variable is initialized in the setup function.
- map – This is the map function of the map reduce job. It receives a text line as input that represent the data of a person. The line is tokenized in an array of strings that holds each feature of the person. The tokenized line is given to the Bayes model to calculate the correlation between this person and the products that the Bayes model holds. The output is a sequence file where the key is the input Object key and the value is the result of the correlation calculation of the model.

2.5.3.6 Bayes Reducer

The classification part of the Bayes Algorithm doesn't have a reducer because is not

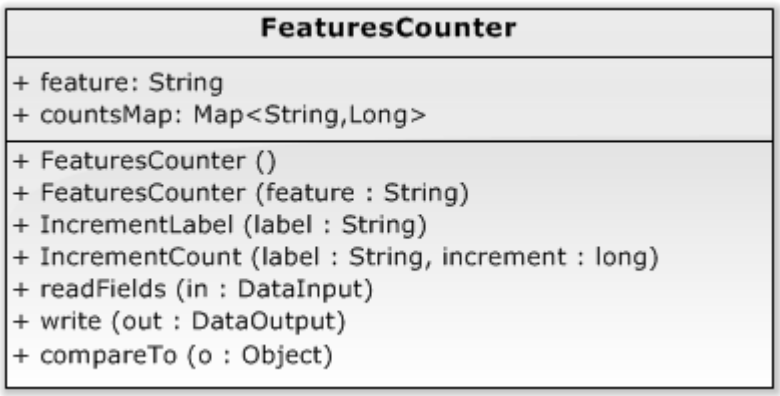
necessary. The only reducer in the algorithm is used in the training part.



- reduce – This overrides the reducer function of hadoop map-reduce reducer class and receives as input the output of the BayesMapper class. The output is a sequence file that contains a collection of key-value pairs. The key is a feature of a person and the output is a FeaturesCounter class that holds a map with the counts of each label for the feature key.

2.5.3.7 Feature Counter

The Features Counter class is used in the Bayes Reducer class to collect all the information of the training data and store it in an efficient way. This class is part of the output of the reducer. This is class is loaded later in the classification step and the Bayes Model is constructed from the data stored in this class.

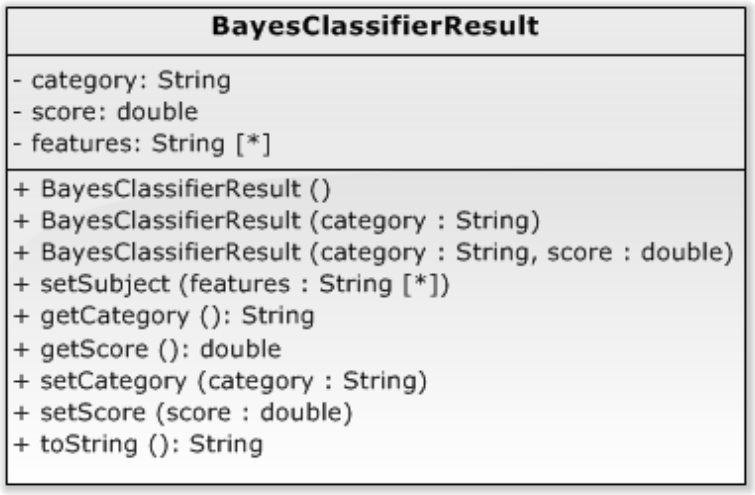


- feature – The feature is being counted for.
- countsMap – Stores the counts of products found with this feature.
- FeaturesCounter – The default constructor. Initializes the countsMap object. The second constructor receives a feature as parameter. This class is storing the counts for this feature.

- IncrementLabel – Increment the count of the label given as parameter by one.
- IncrementCount – Add to the count of the label given as parameter the amount increment.
- readFields – This function reads the data structures and fields of the class from the data input.
- write – Writes all the data structures and fields to the data output.
- compareTo – Compares two FeaturesCounter objects by the feature variable.

2.5.3.8 Bayes Classifier Result

The Bayes Classification Result class is the output of the BayesClassificationMapper class. It contains the result for a line of input data. The content of the class are the category with greater correlation with the features, the correlation score and the features.



- category – The product with the greater correlation with this features. It is the result of the classification step.
- score – The correlation score between the category and the features.
- features – The array of features that identify a line of input data.
- BayesClassifierResult – The constructor of the class. Accepts a category and a category with its corresponding correlation score.
- setSubject – A setter to set populate the array of features.
- getCategory – Returns the category

- getScore – Returnd the score of the correlation.
- setCategory – Set the category variable.
- setScore – Set the correlation score.
- toString – Prints in a meaningful way the category, score and features.

3 Algorithms

3.1 Overview

DAKA includes several algorithms as compute components. These algorithms are generic so that they can be applied to different data sets. This allows the user to solve different problems without needing to rewrite the algorithms. However, using the algorithms requires an understanding of their purpose, strengths, and weaknesses. By understanding both the MapReduce paradigm as well as the details of the included algorithms, the user can make better decisions about using them efficiently.

3.2 MapReduce Paradigm

3.2.1 Overview

The MapReduce paradigm is designed specifically for processing large data sets. A program in the MapReduce paradigm contains two steps, Map and Reduce. The Map step is applied to each record in the input. The results of the mapping are then sorted into groups. Finally, the Reduce step aggregates each group into the final output.

The strength of the MapReduce model does not come from the steps themselves but rather from the frameworks that manage the execution. Because the Mapper runs on each input and the Reducer runs on individual groups, they can each execute in parallel. This opportunity for parallelization is the reason the MapReduce paradigm is useful. When MapReduce programs are run within a MapReduce framework, the framework handles this parallelization creating time efficiency.

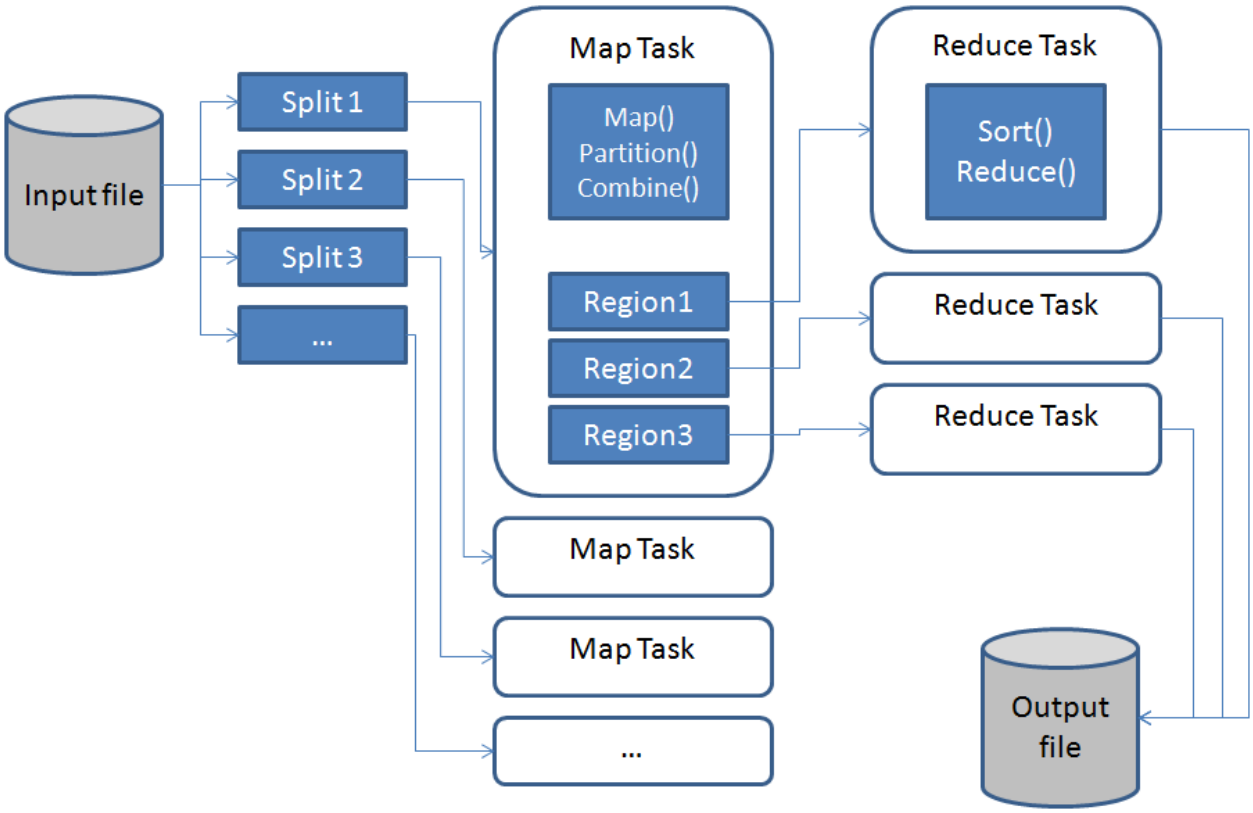


Figure 6 The steps of MapReduce

3.2.2 Map

The Map step is the initial step in a MapReduce program. The Map step consists of a function that takes a single instance of data as input. The function then generates key-value pairs that act as intermediate data. The Map function may emit any number of key-value pairs for any given input. The types of the keys and values do not matter as long as two instances of the key type are comparable.

3.2.3 Sorting

As the intermediate key-value pairs are generated, they are sorted by key. The framework that is managing the execution handles this process. Pairs with the same key are sorted together in preparation for the Reduce step. In a distributed system, the framework also manages the transfer of the values in the pairs to a machine that handles the Reducing of a particular key.

3.2.4 Combine

Combining is an optional step in a MapReduce program. The Combine step is similar to the Reduce step in that it takes a key and a set of values as input. The difference is that the Combine step is not guaranteed to have all values for a given key. This allows the Combine step

to begin before the Map step produces all of the intermediate data. By combining the intermediate data before the Reducer is able to being, the Combiner speeds up execution and reduces the time required for the final Reducing. The use of the Combine step depends on the particular program. In some cases, it can be identical to the Reducer; in others, it is not possible to do any Combining.

3.2.5 Reduce

The Reduce step takes a set of values with a given key and produces the final outputs. The Reduce step assumes that all values with that key are present in the input and it is up to the framework to guarantee this. This step usually performs some sort of aggregation of all of the values. The Reduce function may produce multiple outputs and all of these outputs are members of the final output of the program.

3.3 FP-growth

3.3.1 Overview

Frequent pattern growth, or simply FP-growth, adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases, or Conditional Trees, each associated with one frequent item or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore, this approach substantially reduces the size of the data sets to be searched, along with the "growth" of patterns being examined.

The algorithm performs two scans of the database. The first scan of the database derives the set of frequent items (1-itemsets) and their support counts (frequencies). The set of frequent items is sorted in the order of descending support count. Then an FP-tree is constructed while scanning database a second time.

The construction of a FP-tree occurs in two main steps

1. Determine the frequent items by scanning the data set and discarding the infrequent ones.
2. Scan the data and select one transaction at a time to create the FP-tree.
    a. If it is a unique transaction form a new path and set the counter for each node to one.

   b. If it shares a common prefix itemset, count for common prefix is incremented and nodes for the items following the prefix are created and linked accordingly, thus saving memory.
3. Continue this until each transaction is mapped onto the tree.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree. To mine FP-tree the algorithm starts from each frequent length-1 pattern (as an initial suffix pattern), then constructs its conditional pattern base (a "sub-database," which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern), then construct its (conditional) FP-tree, and performs mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Several rules are used to increase performance of the algorithm:

1) If every transaction containing a frequent itemset X also contains an itemset Y but not any proper superset of Y, then XUY forms a frequent closed itemset and there is no need to search for any itemset containing X but no Y.
2) If a frequent itemset X is a proper subset of an already found frequent closed itemset Y and support count of X equals support count of Y, then X and all of X's descendants in the set enumeration tree cannot be frequent closed itemsets and thus can be pruned.
3) In the depth-first mining of closed itemsets, at each level, there is a prefix itemset X associated with a header table and a projected database. If a local frequent item p has the same support in several header tables at different levels, p can be safely pruned from the header tables at higher levels.

To assist in subset checking, a compressed pattern-tree is constructed to maintain the set of closed itemsets mined so far. The pattern-tree is similar in structure to the FP-tree except that all the closed itemsets found are stored explicitly in the corresponding tree branches. Two-level hash index structure is built for fast accessing of the pattern-tree: the first level uses the identifier of the last item in the itemset as a hash key, and the second level uses the support of the itemset as a hash key.

3.3.2 Parallel FP-growth

Parallel FP-growth (PFP) works by running single-threaded FP-growth on different parts of the database in parallel. It follows following steps to achieve parallelism:

1) Sharding. Dividing database into successive parts and storing the parts on different computers. Such division and distribution of data is called sharding, and each part is called a shard.

2) Parallel Counting. Doing a MapReduce pass to count the support values of all items that appear in database. Each mapper inputs one shard of database. This step implicitly discovers the items' vocabulary. The result is stored in F-list.

3) Grouping Items. Dividing all the items on F-list into Q groups. The list of groups is called group list (G-list), where each group is given a unique group id (gid).

4) Parallel FP-growth. The key step of PFP. This step takes one MapReduce pass, where the map stage and reduce stage perform different important functions:

Mapper: generating group-dependent transactions. Each mapper instance is fed with a shard of database generated in step 1. Before it processes transactions in the shard one by one, it reads the G-list. Mapper it outputs one or more key-value pairs, where each key is a group-id and its corresponding value is a generated group-dependent transaction.

Reducer: FP-growth on group-dependent shards. When all mapper instances have finished their work, for each group-id, the MapReduce infrastructure automatically groups all corresponding group-dependent transactions into a shard of group-dependent transactions. Each reducer instance is assigned to process one or more group-dependent shards one by one. For each shard, the reducer instance builds a local FP-tree and growth its conditional FP-trees recursively. During the recursive process, it may output discovered patterns.

5) Aggregating. Aggregating the results generated in step 4 as final result. This step reads from the output from step 4. For each item, it outputs corresponding top-K mostly supported patterns. In particular, the mapper is fed with pairs in the form of (key = null, value = v, supp(v)), where v is the pattern and sup(v) is the support. For each item a_j in v, it outputs a pair (key' = a_j, value' = v + supp(v)). Because of the automatic collection function of the MapReduce infrastructure, the reducer is fed with pairs in the form (key' = a_j, value' = v + $V(a_j)$), where $V(a_j)$ denotes the set of transitions including item a_j . The reducer just selects from the set top-K mostly supported patterns and out-puts them.

3.4 Naïve Bayesian Classification

3.4.1 Overview

Bayesian classifiers are statistical classifiers. They can predict the probability that a given tuple belongs to a particular class. This classification is based on Bayes' theorem. Bayesian classifiers have exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian Classification is a type of Bayesian classification that works under the *class-conditional independent* assumption. It assumes that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption simplifies the computations involved in the algorithm. It is also the reason that this algorithm is considered "naïve".

3.4.2 Bayes' Theorem

Bayes' Theorem is best understood in mathematical terms. Let ***A*** be a data tuple. ***A*** is considered to be "evidence". ***A*** is many measurements made to a set of *n* attributes. Let H be a hypothesis that says: ***A*** belongs to a specified class C. We want to determine P(H|***A***), the probability that the hypothesis H is true given the observed data tuple ***A***. Another way to say it is that we want to know the probability that tuple A belongs to class C, given that we know the attribute description of ***A***.

P(H|**A**) is the *posterior probability* of H conditioned on **A**. For example, data tuples processed by DAKA may be constrained by the attributes *age* and *income*. Let's say **A** is a 30 years old customer with an income of $50,000 and the hypothesis H is that the customer will buy a whole life policy. Then P(H|**A**) gives the probability that the customer will buy a whole life insurance given that we know the age and the income of the customer.

In contrast, *P*(H) is the *prior probability* of H. Using our example, is the probability that any given customer will buy a whole life insurance regardless his income or age. *P*(**A**) is the prior probability of **A**. The probability that a customer from our data set is 30 years old and earns $50,000.

Bayes' theorem use this concepts to provide a way of calculating the posterior probability, *P*(H|A), from *P*(H), *P*(**A**|H) and *P*(**A**).

$$P(H|\boldsymbol{A}) = \frac{P(\boldsymbol{A}|H)P(H)}{P(\boldsymbol{A})}$$

Another way to understand the relationship demonstrated by Bayes' theorem is through a Venn diagram. The Venn diagram in Figure 7: Probabilities of events A and B7 shows the probabilities of events A and B occurring. The area of the left circle represents the probability that event A occurs. Similarly, the area of the right circle represents the probability that event B occurs. The overlapping area shows the probability that both events occur.

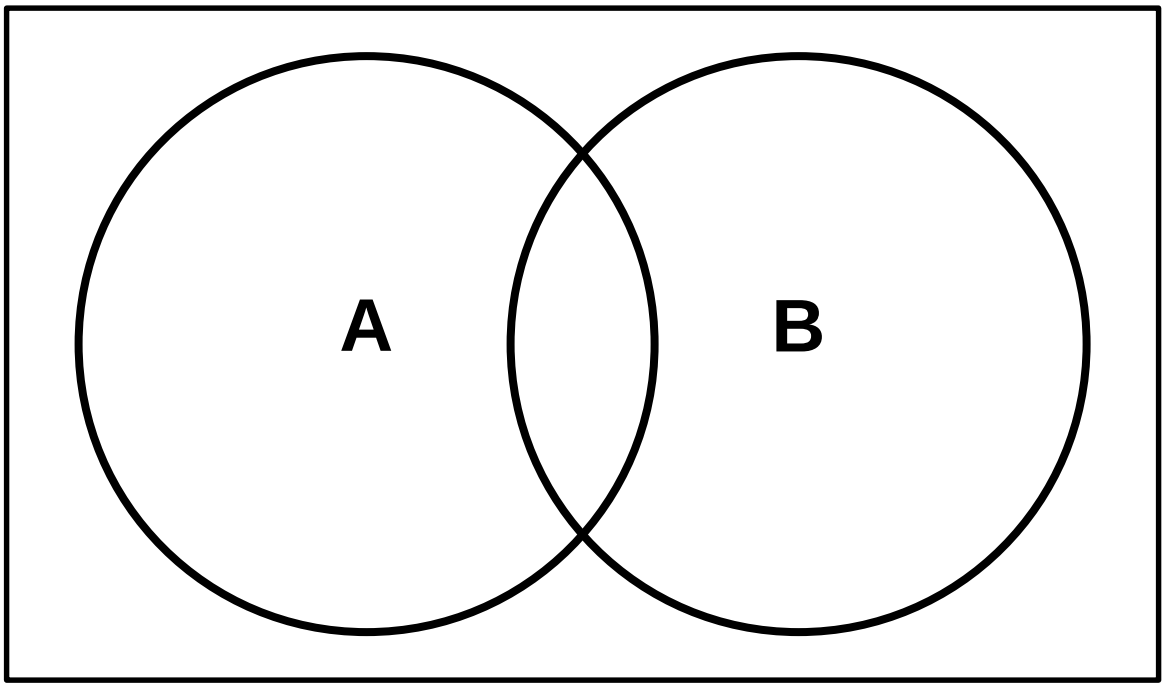


Figure 7: Probabilities of events A and B

Typically, the surrounding rectangle represents the entire realm of possibility. However, since it is given that event B has occurred, the probability of A occurring is equal to the portion of A and B occurring over the total area of B. Another way to formulate the probability of A and B occurring is the probability of B given A multiplied by the probability of A. Therefore dividing this product by the probability gives the probability of A given B.

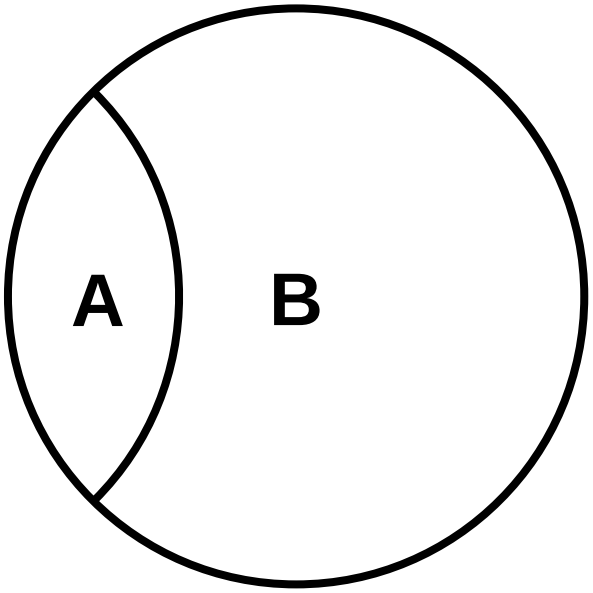


Figure 8: Probabilities of events A and B given B

3.4.3 Classification

The naïve Bayesian classifier works as follows:

1. Let X be a training set of tuples and their associated class labels. Each tuple is represented by a vector, $\boldsymbol{A} = (\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}, \ldots, \boldsymbol{a_n})$ depicting n measurements made on the tuple from n attributes, respectively, $X_1, X_2, \ldots, X_n$.

2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{A}$, the classifier predicts that $\boldsymbol{A}$ belongs to the class having the highest posterior probability.
The naïve Bayesian classifier predicts that the tuple $\boldsymbol{A}$ belongs to the class C_i, only if

$$P(C_i|\boldsymbol{A}) > P(C_j|\boldsymbol{A}) \text{ for } 1 \leq j \leq m, j \neq i.$$

Thus, we maximize $P(C_i|\boldsymbol{A})$. The class C_i for which $P(C_i|\boldsymbol{A})$ is maximized is called the *maximum posteriori hypothesis*. Using Bayes' theorem:

$$P(Ci|\boldsymbol{A}) = \frac{P(\boldsymbol{A}|Ci)P(Ci)}{P(\boldsymbol{A})}$$

3. As $P(\boldsymbol{A})$ is constant for all classes, it is only necessary to maximize $P(\boldsymbol{A}|Ci)P(Ci)$. If the class' prior probabilities are not known, then we assume that the classes are equally likely, and we would maximize $P(\boldsymbol{A}|C_i)$. Otherwise, we maximize$P(\boldsymbol{A}|Ci)P(Ci)$.

The classes' prior probabilities could be calculated by $P(Ci) = \frac{|Ci,D|}{|D|}$ where $|C_{i,D}|$ is the number of training tuples of class C_i in D, where D is our dataset.

4. We estimate the probabilities $P(\boldsymbol{a}_1|C_i)$, $P(\boldsymbol{a}_2|C_i)$, …, $P(\boldsymbol{a}_n|C_i)$ from the training tuples. For each attribute we calculate in different way depending if it is continuous-valued or not. To calculate $P(\boldsymbol{A}|C_i)$, we consider:

a. If the attribute is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D.

b. If the attribute is continuous-valued, then we need to calculate its mean and standard deviation. It is assumed that the attribute it is going to have a Gaussian distribution.

$$g(a, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(a-\mu)^2}{2\sigma^2}}$$

Therefore,

$$P(x_k|C_i) = g(a_k, \mu_{Ci}, \sigma_{Ci}).$$

μ_{Ci} is the mean and σ_{Ci} is the standard deviation for the attribute A_k for training tuples of class C_i. With this two quantities we can plug to the equation to estimate our $P(x_k|C_i)$.

5. To predict the class label of $\boldsymbol{A}$, $P(\boldsymbol{A}|Ci)P(Ci)$ is evaluated for each class C_i.The classifier predicts that the class label of tuple $\boldsymbol{A}$ is the class C_i if and only if

$$P(\boldsymbol{A}|Ci)P(Ci) > P(\boldsymbol{A}|Cj)P(Cj) \qquad \text{for } 1 \leq j \leq m,\ j \neq i$$

In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers; however, in practice this not always the case because we assume the class conditional independence.

We estimate $P(x_k|C_i)$ as the product of the probabilities $P(x_1|C_i)$, $P(x_2|C_i)$, …, $P(x_n|C_i)$,using the assumption of the class conditional independence. But, what happens if the probability of one of the $P(x_k|C_i)$ is zero? This could happen if we don't have training tuples that represent a class. A cancel probability would cancel the effects of the other probabilities. To solve this we apply the *Laplacian correction*.

4 Summary

4.1 Advantages of Design

The component-based architecture chosen for DAKA is ideal because it allows for maximum separation of concerns. This allows each component to work with little or no dependencies on other components. Therefore, users of the framework can selectively use components to fit their needs.

By making use of the Hadoop framework, DAKA gains the ability to scale. Since Hadoop is an established open source software project, resources are readily available. This includes the software itself, documentation on use, as well as community support. The strengths of Hadoop, reliability and scalability, are incorporated into DAKA automatically.

Since DAKA is component-oriented and based on Hadoop, users can easily add their own components. These additional components could be integrations with databases or implementations of other algorithms. This extensibility allows the users to use DAKA to fit their individual purposes.

4.2 Disadvantages of Design

Since DAKA relies heavily on Hadoop, it is vulnerable to any issues with Hadoop. These issues could be defects or difficulties with deploying Hadoop. Because Hadoop is an actively maintained project, these issues should be limited. Another disadvantage is the dependence on multiple machines. Because DAKA and Hadoop scale by distributing work across machines, a cluster of machines is necessary to handle large amounts of data. Maintaining this cluster adds overhead that would not be present in a single machine system.

4.3 Design Rationale

DAKA's design allows for extensibility and reliability. Building on Hadoop allows DAKA to scale reliably for problems with large data sets. The component-based architecture allows the user to add or use functionality as needed.

5 Requirements Traceability Matrix

5.1 Traceability by Requirement

Non-functional requirements are labeled “NFR” to denote that there are no explicit software components designed to specifically satisfy them. These requirements have cross-cutting concerns in which many different components and environmental factors address.

Requirement	Design Component
3.1.1	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.2	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.3	2.5.2.1, 2.5.2.2, 2.5.2.3, 2.5.2.4, 2.5.2.5, 2.5.2.6, 2.5.3.1, 2.5.3.2, 2.5.3.3, 2.5.3.4 , 2.5.3.5
3.1.4	NFR
3.1.5	NFR
3.2.1	2.4.2.1
3.2.2	2.4.2.1
3.2.3	2.4.2.1
3.2.4	2.4.2.1
3.3.1.1	2.4.3.1
3.3.2.1	2.4.3.1

3.3.2.2	2.4.3.1
3.3.2.3	2.4.3.1
3.3.2.4	2.4.3.1
3.3.2.5	2.4.3.1
3.3.3.1	2.4.3.1
3.3.3.2	2.4.3.1
3.3.3.3	2.4.3.1
3.4.1	2.5.2, 2.5.3
3.4.2	2.5.2, 2.5.3
3.4.3	2.5.2, 2.5.3
3.5.1	2.3.2.1
3.5.2	2.3.2.1
3.5.3	2.3.2.1
3.5.4	2.3.2.1
3.5.5	2.3.2.1
3.5.6	2.3.2.1

5.2 Traceability by Design Component

Design Component	Requirement
2.3.2.1	3.5.1, 3.5.2, 3.5.3, 3.5.4, 3.5.6

2.4.2.1	3.2.1, 3.2.2, 3.2.3, 3.2.4
2.4.3.1	3.3.1.1, 3.3.2.1, 3.3.2.2, 3.3.2.3, 3.3.2.4, 3.2.2.5, 3.3.3.1, 3.3.3.2, 3.3.3.3
2.5.2	3.4.1, 3.4.2, 3.4.3
2.5.2.1	3.1.1, 3.1.2, 3.1.3
2.5.2.2	3.1.1, 3.1.2, 3.1.3
2.5.2.3	3.1.1, 3.1.2, 3.1.3
2.5.2.4	3.1.1, 3.1.2, 3.1.3
2.5.2.5	3.1.1, 3.1.2, 3.1.3
2.5.2.6	3.1.1, 3.1.2, 3.1.3
2.5.3	3.4.1, 3.4.2, 3.4.3
2.5.3.1	3.1.1, 3.1.2, 3.1.3
2.5.3.2	3.1.1, 3.1.2, 3.1.3
2.5.3.3	3.1.1, 3.1.2, 3.1.3
2.5.3.4	3.1.1, 3.1.2, 3.1.3
2.5.3.5	3.1.1, 3.1.2, 3.1.3

6 Appendices

6.1 References

Borgelt, Christian. n.d. "An Implementation of the FP-growth Algorithm." http://www.borgelt.net/papers/fpgrowth.pdf.

Borthakur, Dhruba. n.d. "HDFS Architecture Guide." *Apache Hadoop.* https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

Dean, Jeffery, and Sanjay Ghemawatt. 2004. "MapReduce: Simplified Data Processing on Large Clusters ." *Sixth Symposium on Operating System Design and Implementation,.*

Han, Jiawei, Micheline Kamber, and Jian Pei. 2012. *Data Mining: Concepts and Techniques.* Amsterdam: Elsevier/Morgan Kaufmann.

Internet Pipeline, Inc. n.d. "About iPipeline." *iPipeline.* http://ipipeline.com/company/about-ipipeline.php.

The Apache Software Foundation. n.d. *Hadoop Home Page.* http://hadoop.apache.org/.