UPPSALA
UNIVERSITET

IT 18 004
Examensarbete 15 hp
Februari 2018

Investigating the scaleability of analyzing and processing RDBMS datasets with Apache Spark

Ferhat Yusuf Bahceci

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Investigating the scaleability of analyzing and processing RDBMS datasets with Apache Spark

Ferhat Yusuf Bahceci

At the Uppsala Monitoring Centre (UMC), individual case safety reports (ICSRs) are managed, analyzed and processed for publishing statistics of adverse drug reactions. On top of the UMC's ICSR database there is a data processing tool used to analyze the data. Unfortunately, there are some constraints limiting the current processing-tool along with that the amount of arriving data to be processed grows at a rapid rate. The UMC's processing system must be improved in order to handle future demands. In order to improve performance various frameworks for parallelization can be used. In this work, the in-memory computing framework Spark was used for parallelization of one of the current data processing tasks. Local clusters for running the new implementation in parallel was also established.

Handledare: Jonas Ahlkvist
Ämnesgranskare: Olle Gällmo
Examinator: Justin Pearson
ISSN: 1650-8319, IT 18 004
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

1.1 Scope

The main goal of this thesis was to study how parallel computing could be used for processing and analyzing individual case safety reports (ICSRs) [10] and compare the parallel solution to the existing sequential solution. This work was limited by studying and improving a single data analyzing process in the Uppsala Monitoring Centre (UMC) data processing system [2]. Prior to this work, a number of problems with the current implementation of this process had already been identified. Unnecessary database transactions are made for each single ICSR dataset arriving to the database. The SQL server is currently used for the database management required for analyzing the ICSR data today. Since a database transaction consists of a chain of SQL commands [6], all the intermediate results for an ongoing process are stored back to disk between the processing steps. To the hard drive and read from it until the process terminates. Spark offers a different architecture and works around these issues 4.

1.2 Background

This research is a study of the Spark framework as a choice of processing tool [5]. By using Spark for processing and analyzing ICSRs, how quickly would the UMC be able to analyze their entire dataset? Will it be more scaleable and more time efficient to use a Spark based solution?

There are three available and optional cluster managers provided for the distributed in-memory computing framework. One of them is the Hadoop Distributed File System (HDFS) Yet Another Resource Negotiator (YARN) which is a resource distributor to the HDFS [8]. Being a resource distributor means that it is possible to deploy and run Spark containers on the HDFS cluster. Spark reads its input data from the HDFS and writes its result back to it 5.5. Keeping the logic to execute closer to the data might be a very good idea for achieving shorter run times.
HDFS is is distributed data storage and was the actual reason for developing Spark with the goal to achieve in-memory batch-processing and they are both derived from the Apache foundation.

1.3 Purpose

On top of the World Health (WHO) Organization ICSR database called VigiBase there is a data analyzing and processing tool named VigiBase [11] ICSR Processing System (VIPS) [1]. The amount of data to be processed in VigiBase grows at a rapid rate and the processing system VIPS must be improved in order to be able satisfy future demands. Unfortunately, there is not only concerns with VIPS but also with the database VigiBase. Since the ICSR data to be process are resided on a single server, the amount of tasks that can be handled are queued up.

The UMC's current technology are not able to scale out the data and operate on it in parallel. Another issue is that the current solution saves all the intermediate results for an ongoing process back to the hard drive, which leads to spending unnecessary time on Input/output (I/O) operations.

All these issues gives terrible run times and an improvement has to be made since the number of reporters of individual case safety reports (ICSRs) are not going to decrease but only increase. Spark could be an interesting and more scaleable alternative than the current solution. It is however important to remember that Spark is meant to be used for reading and writing its data from a distributed data storage since Spark itself is a distributed analytics and processing tool.

2 System overview

2.1 Description of the processing tool VIPS

The system that stores information related to individual case safety reports (ICSRs) using a number of databases is called VigiBase. VigiBase ICSR Processing system (VIPS) is the name of the processing system built on top of VigiBase. VIPS was constructed for mainly storing ICSR reports but also for analyzing and processing them for publishing statistics and other analytics. VIPS is still under active development. The VIPS system architecture consists of numerous implemented components and has two main requirements [1]:

1. Maintain an efficient production process which means that the solution should especially be stable, reliable and high performing.

2. Upholding an increased quality by focusing on trace-ability, maintainability, processing quality and monitoring.

2.2 VIPS complications and issues

An Individual Case Safety Report (ICSR) is sent as a message through various services in order to be processed. The services perform different types of processing such as data extraction, conversion, transformations and reductions on these ICSR reports.

Issues with the use of a Relational Database Management System (RDBMS) for this kind of work that are slowing down the processing time [1]:

1. The general downside with the current analyzing and processing solution VIPS is that the system end up doing a lot of SQL work for every single report that in turn slows down the overall processing time. A different architecture where the data are resided distributed but also with a different processing algorithm that could keep intermediate data in memory, close to where it will be processed might give processing speedups.

2. Another problem is due to database isolation guarantees. VigiBase internally uses a locking strategy to prevent multiple concurrent transactions from seeing each other's pending changes (isolation in Atomicity Consistency Isolation Durability). This locking strategy can affect concurrent transactions in terms of performance such as throughput and can potentially cause deadlocks. Deadlocks lead to retries, that reduces the performance because it means that a message has to be processed more than once before the results can be successfully committed to the database.

3. Processing ICSRs requires that VigiBase saves all intermediate results to SQL server which leads to the lack of database scale-ability. A RDBMS like SQL server has strong consistency guarantees which is provided at the cost of reduced scale-ability. In other words, you cannot simply scale a RDBMS horizontally over multiple servers unless you share the entire database. In which case, it will end up with a different set of problems.

Since a single server has practical limits to the number CPU cores it can support and Input/Output Operations Per Second (IOPS) it can handle, there is an inherit limit to the performance of the whole system. Even if you would add an infinite number of servers for processing the reports.

The goal of the work presented in this report is to investigate if a different architecture based on distributed data, scaleable parallel processing and keeping intermediate results in memory might be a good fit for the UMC's ICSR analyzing and processing needs. [1]

3 The data analysis process to re-implement using Spark

3.1 The IC value

The data analysis process that was chosen to be re-implemented for in Spark is a mathematical calculation called the Information Component (IC) [2]. The IC value contains a measure of the dis-proportionality between the observed and the expected reporting of a drug-adverse drug reaction pair (drug-ADR pair). The IC value shows the quantitative dependency between the ADR and the drug. The higher the value of the IC, the more the combination stands out from the background. The IC value does however not imply the causality of a potential adverse reaction caused by a drug. Newly arrived individual case safety reports (ICSRs) may cause the IC value to either increase or decrease. The properties required for a single IC calculation are extracted from the UMCReports (which are UMC internally converted ICSRs) that are submitted to the World Health Organization (WHO) global ICSR database (VigiBase). [2]

- An IC value greater than zero indicates that a particular drug-ADR pair is reported more frequently than expected.

- An IC value smaller than zero indicates that the drug-ADR pair is reported less frequently than expected.

3.2 Estimation of the IC value

An IC value is estimated by examining a certain drug (MedicinalProduct) and the adverse drug reaction (ReactionMeddraPt) experienced from using this drug. These are the values required for calculating a single IC value for a specific drug-adverse drug reaction pair (drug-ADR-pair) and they can be queried from each UMCReport respectively arriving to the VigiBase database. A single patient can however have many drug-ADR pairs if the patient consumes many drugs and experiences many adverse drug reactions from these drugs. [2]. For example the drug-ADR-pair→(X,Y)

1. Number of patients **USING** the drug X and that **DO EXPERIENCE** the adverse drug reaction Y.
2. Number of patients **USING** the drug X and that **DO NOT EXPERIENCE** the adverse drug reaction Y.
3. Number of patients that **DO NOT USE** the drug X and that **DO EXPERIENCE** the adverse drug reaction Y.
4. Number of patients that **DO NOT USE** the drug X and that **DO NOT EXPERIENCE** the adverse drug reaction Y.

3.3 Calculation of the estimated IC value

An example of a single IC calculation for the drug-ADR pair→(MedicinalProduct: Enalapril, ReactionMeddraPt: Coughing), Figure 1.

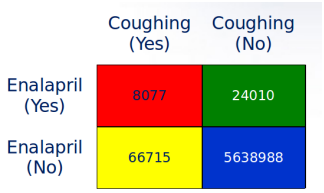

Figure 1: Values required for calculating the IC value for the drug-ADR pair→(Enlapril, Coughing) [2].

When the values shown in Figure 1 are retrieved by database transactions, the IC value can be calculated.

1. Total amount of UMCReports in the Uppsala Monitoring Centre database, N_{Tot} = 8077 + 66715 + 24010 + 5638988 = **5 737 790**
2. The total number of reports on the specific drug-ADR pair, N_{Comb}. UMCReports with combination , N_{Comb} = **8077**

3. The total number of reports on the ADR term, N_{Adr}.
   UMCReports with coughing, N_{Adr} = 8077 + 66175 = **74 792**. I.e 1.3% of N_{Tot} of all the UMCReports in VigiBase where coughing was reported.

4. The total number of reports on the Drug term, N_{Drug}.
   UMCReports with Enalapril, N_{Drug} = 8077 + 24010 = **32087**.

How common is the usage of the drug Enalapril in UMCReports based on the information from Figure 1?

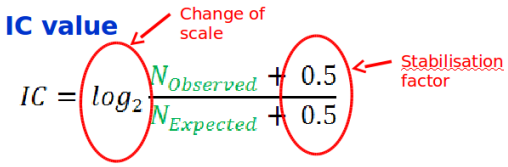

Figure 2: Formula for an IC calculation [2].

- Observed value: $N_{Observed}$ or N_{Comb} = 8077
- Expected value: $N_{Expected} = (N_{Adr}/N_{Tot})*(N_{Drug})$ = 0.013*32 087 = 417.1
- Insertion into the IC formula, Figure 2 gives the following answer: **4.27**

Coughing seems to be more frequently reported in Enalapril reports than reports in general. [2]

4 General information about Spark

Apache Spark

Spark is a powerful processing engine, a cluster framework developed at the University of California, Berkeley in 2009. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. The framework was open sourced in 2010 and has been the largest open source project in data processing ever since. Since Spark is a polyglot framework it means that the framework provides high-level API's in R, SQL, Java, Scala and Python. The main purpose of Spark was to provide a processing framework for very large quantities of data built around speed, ease of use as well as for sophisticated analytic. Spark is currently the fastest big data processor and holds the world record for large-scale on disk sorting. [9]

Spark components

The IC application was written in Scala together with Spark's own SQL i.e JVM based. A closer approach on Spark's database transactions had to be made and

explained for understanding how the framework implements SQL functionality. Spark is a processing engine with only one purpose, analyzing and processing big datasets. A Relational Database Management System (RDBM) table or any other text format consisting of structured data can be loaded into Spark as an abstraction called DataFrame. A DataFrame is an API for building a relational query plan that the Spark's Catalyst optimizer can execute. Most of the SQL functionality comes due to the Catalyst optimizer.

4.1 Spark's Catalyst optimizer applies SQL functionality

The Catalyst optimizer comes with a general framework for transforming trees by primarily leveraging the functional programming constructs of Scala's pattern matching. The trees, are the logical plans that are constructed and used to perform runtime analysis, planning, optimization and code generation. A set of SQL queries are basically represented in these trees. The optimizer has two primary goals [7]:

1. Adding new optimization techniques easily.
2. Allowing external developers to extend the optimization.

Spark SQL uses this Catalyst's tree transformation framework in four phases:

4.2 Analyzing and delivering a logical plan to resolve the references

The first phase of the optimizer involves the studying of all the SQL queries that are made in the program code, creating an unresolved logical plan by constructing a tree out of the queries that are wished to be performed. The logical plan is unresolved since the columns referred may not exist or may be of the wrong data type, just like a visualization that only contains the references and not the actual data, for example a table VIEW in SQL. The logical plan is resolved by the Catalog object i.e. a component in Spark's catalyst optimizer that connects the database transactions with the physical data source, Figure 3. The data source in Spark is the DataFrame where all the structured data are loaded into. [7]

4.3 Logical plan optimization

The second phase involves optimizing the logical plan by applying the standard rule provided by the Catalyst's transformation framework. The standard rule includes constant folding, predicate push downs, null propagation and Boolean expression simplification. Optimization's are used to improve poorly designed schema data that are recognized in the DataFrame, Figure 4. For example undefined or unknown properties that have no value assigned to them are all NULL: ed in the optimization phase. All columns are made atomic. The optimization phase ensures that there is only one single value for each column row

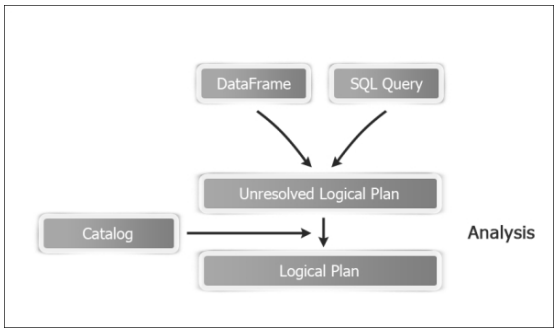

Figure 3: Phase 1 of the Catalyst optimizer. [7]

of the output resilient distributed dataset (RDD) created when Spark's catalyst optimizer is called on a DataFrame. RDD is the actual representation of Spark manageable data that enables parallelism. [7]

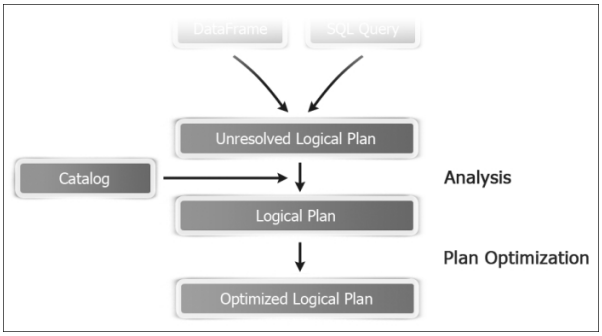

Figure 4: Phase 2 of the Catalyst optimizer. [7]

The standard rule is the collection of compiler optimization techniques applied by the Spark Catalyst during the logical plan optimization phase.

- Constant folding is the process of evaluating constant expressions at compile time rather than computing them at run-time. For example X = 5 + 1 would be set to X = 6 during compile time. [7]
- Predicate push downs is the basic idea of processing expressions as early in the execution plan as possible. The purpose of predicate push downs is to increase the performance of the database transactions. [7]
- Null propagation is the optimization of the dataset by nulling unknown values that are present in the data set. Null propagation provides a cleaner relation schema rather than a poor one. [7]
- Boolean expression simplification are expressions that represent the combination of logic circuits. The simplification reduces an expression to

another expression that are equivalent with each other. It simply takes a Boolean expression and reduces it to another equivalent but with fewer operators. [7]

4.4 Physical planning

Phase number three is where the optimized logical plan is managed by Spark SQL for generating one or more physical execution plans. The costs of generating each of these physical execution plans are measured and only one physical plan is selected based on the explored generating costs. In this phase, all the possible candidates of execution plans are generated. The execution plan with the most effective execution plan is then selected and generated as the final physical plan, Figure 5. [7]

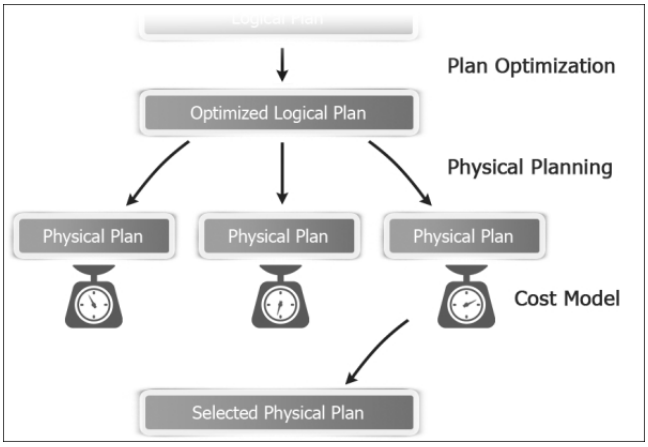

Figure 5: Phase 3 of the Catalyst optimizer. [7]

4.5 Code generation

The last and the final phase of the Catalyst optimization involves generating Java byte-code for the selected execution plan. This is accomplished by a feature called Quasi quotes provided in Scala. The Java byte-code is the actual code for the database transactions that are going to run on the machines across the cluster. The code generation is the final point where a database transaction is fulfilled and translated into Spark's core concept, Resilient Distributed Datasets (RDDs). Now that the retrieved data is in Spark manageable format it can be used to be operated on in parallel, Figure 6. [7]

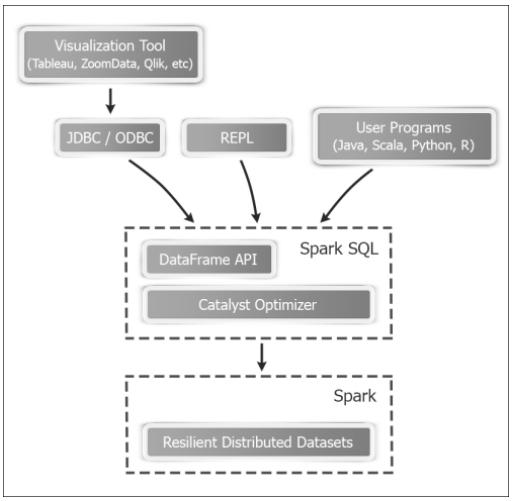

Figure 6: Overview of Spark's Catalyst optimizer. [7]

4.6 Resilient distributed dataset - the core concept in Spark

A DataFrame is comparable with a RDBMS table. When structured data are loaded into Spark, the set of data are all gathered in one unit called DataFrame. When database transactions are made on the DataFrame, the Spark Catalyst optimizer converts this DataFrame to Spark's own representation of data. A RDD is essentially the Spark representation of a set of data. A RDD is the basic abstraction in Spark representing an immutable, partitioned collection of elements that can be operated on in parallel. The RDD is a distributed memory abstraction that lets the developer perform in-memory computations on large scalable clusters in a fault-tolerant manner. All the data records held within a RDD can be stored on different partitions across a cluster, Figure 7. These partitions enable the rearrangement of the computations by optimizing the data processing through a parallel computation framework with a higher-level programming interface. The RDDs are partitioned to a determined number of partitions. The partitions are considered as logical chunks of data. All these

sliced and distributed chunks of data together constitute the total dataset RDD, Figure 7. The total RDD is not internally divided. The purpose of this logical division is for processing only and for making it possible to operate on the RDD in parallel. The partitions of the RDD are therefore the units that are enabling parallelism in Spark. [7]

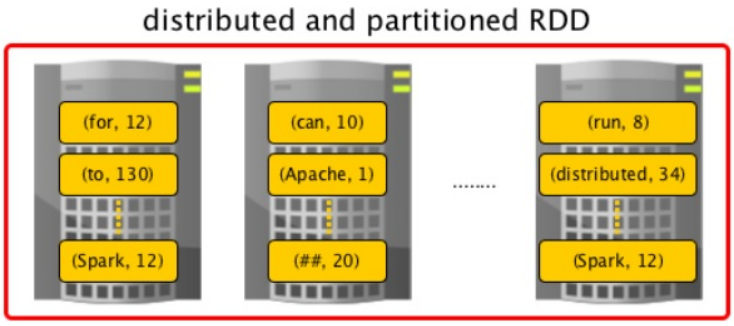

Figure 7: The RDD partitions are distributed across multiple machines, and they are all together the unit RDD. [7]

Additional traits of RDDs [7]:

- In-Memory:
  All the containing data inside of a RDD are stored in in-memory as much size and long time as possible. Thereafter spilled to secondary storage. This means that many unnecessary Input/Output (I/O) operations can be avoided by being prevented by Spark's in-memory feature.
- Immutable or Read-Only:
  Once the RDD is created, it cannot be changed since it its immutable. The RDD can only be transformed using transformations to a new RDD.
- Lazy evaluated:
  The data inside of a RDD is not available or transformed until some action is executed on the RDD. Data values are retrieved after an execution of an action.
- Cache-able:
  It is possible to dedicate memory to the Spark application so that all the data can be held in a primary and persistent storage like in Random Access Memory (RAM) but also in virtual memory.
- Typed:
  All values in a RDD have types, RDD[(Int, String)], RDD[Long], etc.
- Partitioned:
  All the containing data of a RDD are sliced into logical chunks and distributed across nodes in a cluster.

4.7 RDD operations

RDDs have two types of operations, actions and transformations. Actions returns values while transformations return a new RDD. A transformation call does not evaluate anything. A transformation takes a RDD and returns a reference by a given pointer to that RDD in a changed form. The RDDs are lazy evaluated. All the data processing queries are computed at the time when an action function is called on a RDD object and the result value is thereby returned. The transformations are only references of the most native RDD and nothing are really evaluated until an action is made upon the transformed RDD. [7]

4.8 Shuffling

The process of redistributing data across partitions are called shuffling or repartitioning. Shuffling is the process of data transfer between execution phases in Spark. Shuffling may or may not cause moving data across JVM processes or even over the wire between executors on separate machines on a distributed cluster. Shuffling should therefore be avoided since it is better to leverage existing partitions than redistributing data between a parallel system's executors. [7]

4.9 Jobs

A job is a top-level computation that computes the result of an action. A job is equivalent to computing partitions of a RDD that an action has been made on. The amount of partitions that a job contains depends on the type of the job's stage. There are two types of job stages, ResultStage and ShuffleMapStage. [7]

4.10 Stages

Sequences of tasks to execute are called stages, Figure 8. A stage is a step in a physical execution plan. A physical unit of an execution. A set of parallel tasks that are executed for each partition of a RDD. Each partition of the RDD computes partial results of a function executed as a part of a Spark job that are triggering stages. [7]

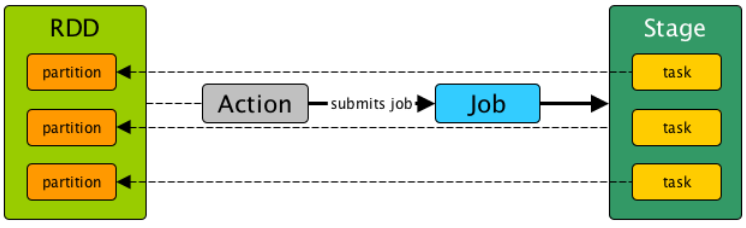

Figure 8: RDD stage initialization [7].

4.11 Spark actions triggers a stage

All Spark applications involves jobs that are triggered by actions. A Spark application can consist of several jobs. These jobs are executed in sequences of tasks called stages. A Spark job is simply a computation which is sliced or partitioned into one or more stages, Figure 9. A stage is usually logically divided into several stages. Each stage is uniquely assigned with a stage ID for keeping track of the stages. A stage can only operate on the partitions of a single RDD. A divided stage is associated with many other sub-stages. Submitting a stage can therefore trigger the execution of a series of sub-stages. [7]

1. ShuffleMapStage:
   The intermediate sub-stages of a total stage. ShuffleMapStage is responsible for producing intermediate results for an ongoing stage. This stage can be compared to the Map phase of the MapReduce paradigm.

2. ResultStage:
   The final sub-stages that evaluates the final result of a total stage. This stage can be compared to the Reduce phase of the MapReduce paradigm.

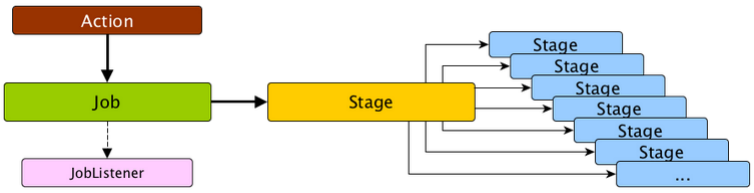

Figure 9: Stage initialization overview [7].

4.12 Tasks

A task is a command and is the last in the execution line in Spark. The smallest individual unit of execution that represents an execution of a partition of a RDD, Figure 10. Each task is managed by a CPU core. [7]

A task is a computation on a data partition in a stage of a job. This means that a task can only belong to one stage as well as only operating on a single partition of a RDD for each task. Before the next scheduled stage can start, all the other tasks in the previous stage must be completed. The tasks are spawned one by one for each stage.

All tasks in Spark are represented by an abstract class called Task that has two concrete implementations [7]:

1. *ShuffleMapTask*, responsible for executing tasks of intermediate stages that are required for the final result. Sub-results or intermediate results of a stage that other stages are based on.

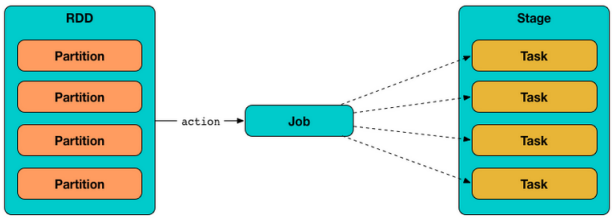

Figure 10: From RDD partitions to respectively tasks [7].

2. *ResultTask*, responsible for executing the tasks of the final stages. When the task has terminated, it sends the task's output results to the driver application. The very last stage in a job consists of several ResultTasks while previous stages are a set of ShuffleMapTasks.

4.13 Directed Acyclic Graph Scheduler

At this point of time some kind of structured data has been loaded into a Spark applications DataFrame. Spark has thereafter translated the DataFrame to its own internal manageable data format called RDD. Now that actions are made on the newly made RDD a job is triggered. A stage is created when a job is submitted. The DAG Scheduler splits a job into a collection of stages. The scheduling layer of Spark is the DAGScheduler. The DAGScheduler consists of some significant concepts which are Jobs→Stages→Tasks and the DAGScheduler is directly coordinated by the SparkContext.

Once an action has been made on a RDD it becomes a job. A job is thereafter transformed into a set of stages. Each stage has a set of tasks to execute and they are respectively scheduled by the TaskScheduler. The DAGScheduler builds up a DAG by strictly applying topological ordering with each stage assigned to a node respectively of the DAG. This means that child nodes in a DAG cannot be executed before the parent nodes. For example, A ShuffleMapStage is an intermediary sub-stage with the only purpose to produce input required for the final ResultStage. The ResultStage can therefore not be executed before the ShuffleMapStage. Mapping means that each sub-stage holding a set of tasks from a logically divided stage is respectively inserted in an execution order into the nodes of a DAG. Each node in a DAG therefore holds sets of tasks to execute by applying topological ordering. These nodes of the DAG are stages that holds specific sets of tasks that are distributed between the clusters executors. [7]

4.14 TaskScheduler

The TaskScheduler is responsible for ensuring the execution of the sets of tasks for each stage that are submitted to it from the DAGScheduler. The TaskScheduler directly communicates with the SparkContext and is also coordinated by the SparkContext. The tasks within a stage are executed concurrently and in

parallel thanks to the features provided by RDDs. The TaskScheduler utilizes each available CPU core there is in the cluster. Diverse partitions of the RDD are spread across the nodes in a cluster. Each task is a work of executing an operation of an assigned RDD partition. The tasks are scheduled by the TaskScheduler. [7]

5 Distributed workloads with Spark

5.1 SparkContext

The entry point to all applications in Spark is the SparkContext. Every Spark program requires a SparkContext and the SparkContext requires a SparkConf. The SparkConf is where all the application settings, parameters and properties are set and passed on to the SparkContext. For example the name of an application in Spark is set in the SparkConf. The SarkContext is responsible for setting up internal services as well as the establishment of a connection to a Spark execution environment, i.e. to an established cluster. The SparkContext acts as the master of the Spark application. The SparkContext represents the connection to a Spark cluster and it provides every functionality needed for creating a RDD on that cluster. [7].

5.2 Driver

The driver is the host of the SparkContext required for a Spark application. The driver talks to a single coordinator called master. The master manages workers in which executors run. The driver communicates with each node in the cluster through a cluster manager. The driver is responsible of dividing a Spark application into several tasks for scheduling them to run on different executors across the cluster. It is up to the cluster manager in what way these tasks are going to be distributed and how resources are going to be shared between each node participating in the cluster. The driver is thereby responsible for coordinating the workers and the overall execution of tasks. The workers in an established cluster are responsible for executing tasks that are assigned to them by the driver. The driver is simply the workload coordinator in a Spark application and it communicates with its workers through a cluster manager that specifies in what way the workloads are going to be distributed but also how the resources are going to be shared between the executors. There exist three possible cluster managers in Spark, one of the options has to be selected. [7].

5.3 Spark engine on clusters

Applications in Spark runs as independent sets of processes that are wide spread on a cluster. The RDD is partitioned and each partition of the RDD is spread across the cluster. Each executor in the cluster operates on a partition of the RDD. Therefore, no locks are required for preventing concurrency errors since all

the executors operates on diverse partitions of the total RDD. The application is all coordinated by the SparkContext object found in the main program and the driver includes the SparkContext, Figure 11. To run a Spark application on a cluster, the SparkContext has to be connected to one of the three cluster managers provided in Spark, i.e Spark Standalone Mode, HDFS YARN and Mesos. A cluster manager is responsible for allocating resources across application processes. Once a cluster manager is connected, the Spark application can begin acquiring executors on the nodes in the cluster. The next step is that the SparkContext sends the application code (JAR or Python files that are passed to the SparkContext) to the executors. Finally, the SparkContext is able to send tasks to its executors for execution. Each executor is managed by CPU core and directly communicates with the SparkContext. The worker nodes in the cluster are workers running computations that are assigned to them by the driver but they also store data for the Spark application. [9].

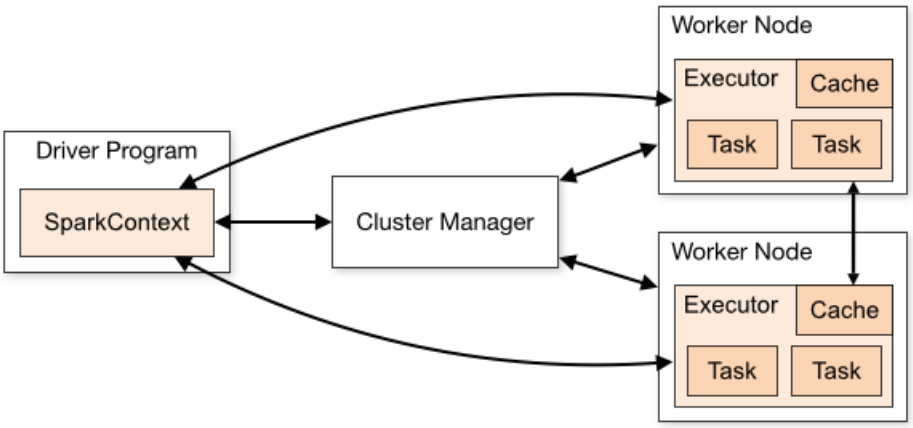

Figure 11: Overview of cluster mode [9].

5.4 Spark's Standalone cluster manager

Spark Standalone cluster is Spark's own built-in clustered environment and it is available in the default distribution of Spark, Figure 11. This cluster manager is the easiest way to run a Spark application in a clustered environment. To install Spark Standalone cluster, a compiled version of the Spark application is placed on each node of the cluster. This means that every node in the cluster requires a Spark installation.

A Standalone cluster in Spark consists of two main components [9]:

1. Standalone Master, which is the resource manager for the Standalone cluster.

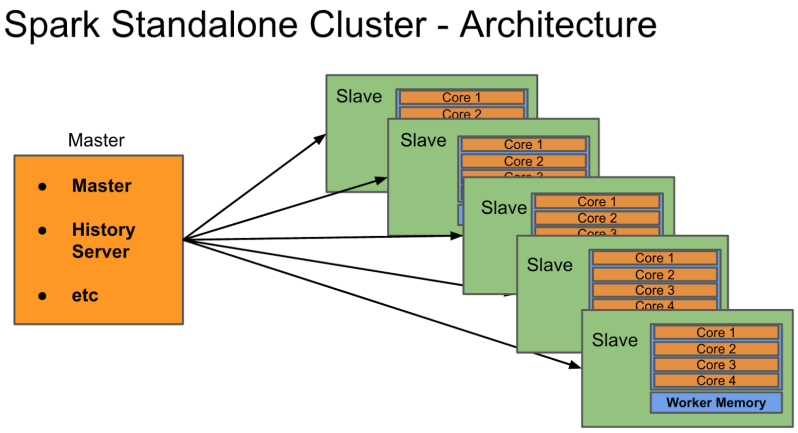

Figure 12: Overview of Spark's Standalone cluster mode [9].

2. Standalone Workers, are the workers, the executors in the Spark standalone cluster that does the execution that are assigned to them by the Master.

In a Standalone cluster, Spark allocates resources to its executors based on the available cores in the cluster. By default, an application will allocate all the possible and available cores existing in the cluster. Each worker node requires to have at least one CPU core assigned to it. Once a Standalone cluster has been deployed, it can be accessed and monitored through a web UI using spark: // master URL. A Standalone cluster can be launched either by manually spawning a master and thereafter workers. One by one manually or just use the provided launch script that launches the whole cluster recursively with a single command. Launching a cluster step-wise [9]:

1. ./sbin/start-master.sh

2. ./sbin/start-slave.sh <master-spark-URL>

Once the master is started, an URL will be printed which is used to connect workers to it. This URL is the Web UI that helps to monitor the state of the cluster by giving information about each worker node within it. Launching a cluster at once:

./sbin/start-all.sh

The cluster establishment basically involves that each node participating in the cluster are defined in the: *spark/conf/slaves* file on the master node. All the IP addresses of the host nodes participating in the cluster are set in this file. Each worker node has its own specific properties that are specified in the *spark/conf/spark-env.sh* file on each node respectively participating in the

cluster as well as the required IP address of the master node is also set in this file for being able to connect to the master of the cluster. Remote connection from the master host to the worker nodes are established by using Secure Shell (SSH). The worker nodes or even called slaves in the cluster has to be specified with certain concurrency properties so that the processes somehow do not manage to steal resources from each other, in terms of Random Access Memory (RAM) and CPU time. The programmer can configure just one single worker node with all the available resources existing (i.e. the CPU cores and the RAM) allocated to just one single worker node. This means that there can exist only one worker node on a host node which is allowed to handle one concurrent tasks of a process for each available CPU core existing on the computer. [9]

5.5 Hadoop YARN cluster manager

Hadoop is an open-source software framework for establishing clusters with distributed data storage management of very large datasets. Hadoop also includes a processing component used for batch-processing data, applying the MapReduce workflow. Hadoop is usually called HDFS. Since the HDFS has issues regarding processing in the way that intermediate results for ongoing processes are saved persistently between the execution phases of a process. Spark was developed to solve this specific HDFS issue by keeping all the data in in-memory as much and as long time as possible before spilling it to disk. [8]

Hadoop Yet Another Resource Negotiator (YARN) is the resource-management platform that are responsible for managing resources that are required for each computing node in the Hadoop cluster. The resource manager required for handling batch-processing, read and written back to the HDFS.

YARN runs processes on a cluster similarly to the way an operating system runs processes on a standalone computer, which is concurrently. It is possible to deploy Spark containers on a HDFS YARN cluster. Resources are assigned to processes that really needs them, not more and not less. Daemons in computing terms are the processes that runs in the background. Hadoop has several such daemons. [8]:

1. The NameNode is the centerpiece of the HDFS file system. It is responsible for keeping track of a directory tree of all the present files kept in the HDFS file system. References to where data are actually kept in the HDFS cluster are hold by the NameNode but not the actual data to be accessed itself. The NameNode communicates with client applications whenever they wish to locate and access a file for the purpose of delete, add, modify, move or even copy a file. The NameNode has to fulfill these application request that are made against the HDFS file system when data is wished to be reached. The NameNode responds successfully to a request by returning a list of relevant DataNode hosts where the searched data resides. Distributed file storage management part, acts as part of master for storage management.

2. The DataNode stores the data in the HDFS. A distributed functional file

system has more than one DataNode and their data are replicated across the nodes in the cluster. This is achieved by allowing the DataNodes being able to communicate with each other in terms of replication. Distributed file storage management part, acts as data file container.

3. The ResourceManager is the Master daemon that is responsible for communicating with the workers in terms of negotiating requesting resources. The ResourceManager are tracking resources on the cluster, and orchestrates the clusters resources to the NodeManagers in the HDFS cluster. The ResourceManager is located on the same host node as the NameNode. Distributed resources required for distributed processing part, acts as part of master for processing.

4. The NodeManager in the HDFS cluster is the worker daemon that launches and tracks processes spawned on the worker hosts. The NodeManager is located on the same host nodes as the DataNodes. For each DataNode there is a NodeManager. This for ensuring that data to be accessed for a process can directly be accessed and be processed at the same host node. The business logic and the data are resided closely next to each other. Distributed processing part, acts as worker.

5. The TaskTracker is responsible for receiving and accepting tasks assigned to it and they are spawned when an application is submitted on top of the HDFS cluster. Every TaskTracker is configured with a set of slots for indicating the amount of tasks a worker node can accept. The TaskTracker is directly triggered by the JobTracker. Distributed processing part, acts as task coordinator.

6. The JobTracker is the master service within the HDFS file system that distributes MapReduce tasks to specific nodes in the Hadoop cluster. Preferably to the DataNodes where the data to be accessed is located and to that node storing the specific data required for executing the process. Distributed processing part, acts as task assigner.

All the properties required for a YARN cluster are defined and configured in the YARN configuration XML files. These files are placed in the same specific directory location on each host participating in the cluster. The master node of the HDFS cluster is defined in the *core-site.xml* file. The file *yarn-site.xml* is used for configuring the behavior of the ResourceManager in the HDFS cluster. The file required for configuring the NameNode and the DataNode respectively is called *hdfs-site.xml* and this file contains the actual configurations varying from each host defined in the cluster. [8]

YARN has two defined resources, vcores and vmemory (virtual cores and virtual memory). Each NodeManager in a YARN cluster keeps track of its own local resources. The NodeManager communicates and alerts its resource configurations to the ResourceManager. The ResourceManager keeps a running total of the cluster's available resources by presenting vcores and vmemory. By keeping track of the total available resources, the ResourceManager is able

to allocate resources as they are requested. The ResourceManager distributes the available resources existing in the cluster to each NodeManager requiring resources. The vcore has a special meaning in YARN, it can be seen as a usage share of CPU cores. [8]

An important concept in YARN are containers which can be considered as a request to hold resources on the YARN cluster. A container holds resources consisting of vcore and vmemory. The NodeManager is able to launch a process called task once a hold has been granted on a host. [8]

An application in YARN is a YARN client program that is made up of one or more tasks. There is for each running application, a special piece of code called an ApplicationMaster which is responsible for coordinating the tasks on the YARN cluster. The ApplicationMaster is the first process spawned after an application starts. An application running tasks on a YARN cluster consists of the following steps [8]:

1. The application starts and talks to the ResourceManager of the cluster.
2. The ResourceManager makes a single container request on behalf of the application.
3. The ApplicationMaster starts running within that container allocated by the ResourceManager.
4. The ApplicationMaster requests subsequent containers from the ResourceManager that is needed to run tasks later on for the whole application process. Those tasks do most of the status communication with the ApplicationMaster.
5. Once all tasks are finished, the ApplicationMaster can finally exit and the last container is de-allocated from the cluster.
6. Final step, the application client exits.

5.6 Similarities and differences between Spark and Hadoop

The two frameworks do different things as well as serving different purposes. Hadoop is actually an infrastructure for distributing datasets across multiple nodes, a distributed file storage management system. Hadoop distributes datasets on multiple nodes and indexes the data so the Hadoop system can keep track of the data. The main goal of Hadoop is actually to store data across a cluster and not to process the data. Spark on the other hand is a tool used for data-processing that should preferably operate on distributed data collections. Spark does not really do any distributed storage.

Hadoop does not only work as a storage component but the Hadoop Distributed File System (HDFS) also includes a processing component applying the MapReduce workflow. Since the data is wide spread in a Hadoop cluster it enables faster data access when it comes to processing. Therefore, HDFS YARN is a cluster manager for deploying different application containers that

the HDFS YARN supports, and one of them is Spark. In a HDFS YARN cluster the executors does not have to waste unnecessary access times for accessing data required for execution of a process since the data is located close to where it will scheduled and executed. No unnecessary loading in and out from any external storage management system into the hard drive of the execution nodes are required. The worker nodes have a shorter distance to loading it in to its RAM for execution which provides faster execution times. Since Spark does not come with its own file management system, it requires to be integrated with one, such as HDFS or some other cloud-based data platform. Spark was primarily developed for the purpose of improving a way of processing data resided on a HDFS, this was actually the spark of Spark. Spark was developed for operating on distributed data collections such as provided by the HDFS YARN. [8]

The processing component in HDFS had to be improved. This was achieved by developing the in-memory computing framework Spark. Spark is generally faster when it comes to processing data compared to the way MapReduce processes data. The MapReduce paradigm works in steps clearly described by this quote:
"The MapReduce workflow looks like this: read data from the cluster, perform an operation, write results to the cluster, read updated data from the cluster, perform next operation, write next results to the cluster, etc. Spark, on the other hand, completes the full data analytic operations in-memory and in near real-time: "Read data from the cluster, perform all of the requisite analytic operations, write results to the cluster, done, Spark can therefore be as much as ten times faster than the MapReduce workflow for batch processing. It can also be up to hundred times faster for in-memory analytics" - Kirk Borne, principal Data Scientist at Booz Allen Hamilton.. [8]

6 Results

6.1 Cluster environment

To be able to draw reasonable conclusions out of the results of the performance measurements that were made, it is important to present all the factors that the results depend on. The run times of a Spark application strongly depends on the available resources in the cluster, that is the network bandwidth, CPU or the amount of RAM assigned per executor. In this subsection, a short description of respectively device participating in the cluster will be provided. All the results in this section is from submitting the IC application in the Spark-Hadoop version called: *spark-2.0.0-hadoop* which is a Spark version including Hadoop. This means that Spark will apply Hadoop file distribution when writing data results back to the hard drive.

Figure 13: Computer information of PC_1, i.e the master host.

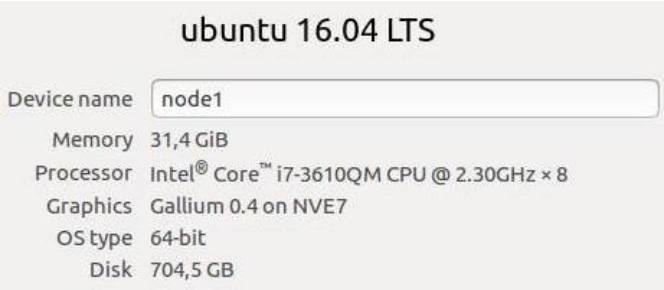

Figure 14: Computer information of PC_2, i.e the worker host.

6.2 Cluster setups and results from Spark submits

Spark's Standalone Cluster - performance measurements and tests

To investigate Spark's scalability, a reasonable approach was to double the amount of worker nodes between each test, i.e. the submit of the implemented Spark application for the assigned process of this thesis, IC calculation.

The hardware used in this experiment was able to assign each worker instance (a worker node in Spark's Standalone cluster mode) with one executor core and 5GB of Random Access Memory (RAM) each. The tests were made for a total of **11 094 UMCReports** on top of a Standalone cluster in client deployment mode.

1. Test 1: Cluster with master and just **one** worker instance (one worker node) on PC_1, *hence* 2^0 *= 1 core.*
2. Test 2: Cluster with master and **two** worker instances (two worker nodes) on PC_1 , *hence* 2^1 *= 2 cores.*
3. Test 3: Cluster with master and **four** worker instances (four worker nodes) on PC_1, *hence* 2^2 *= 4 cores.*

4. Test 4: Cluster with master and **eight** worker instances (eight worker nodes) on PC_1 and PC_2 , *hence* $2^3 = 8$ *cores.*

The tests shown in Figure 15 includes the time required to store the results to the hard drive. All the IC values are calculated and saved persistently.

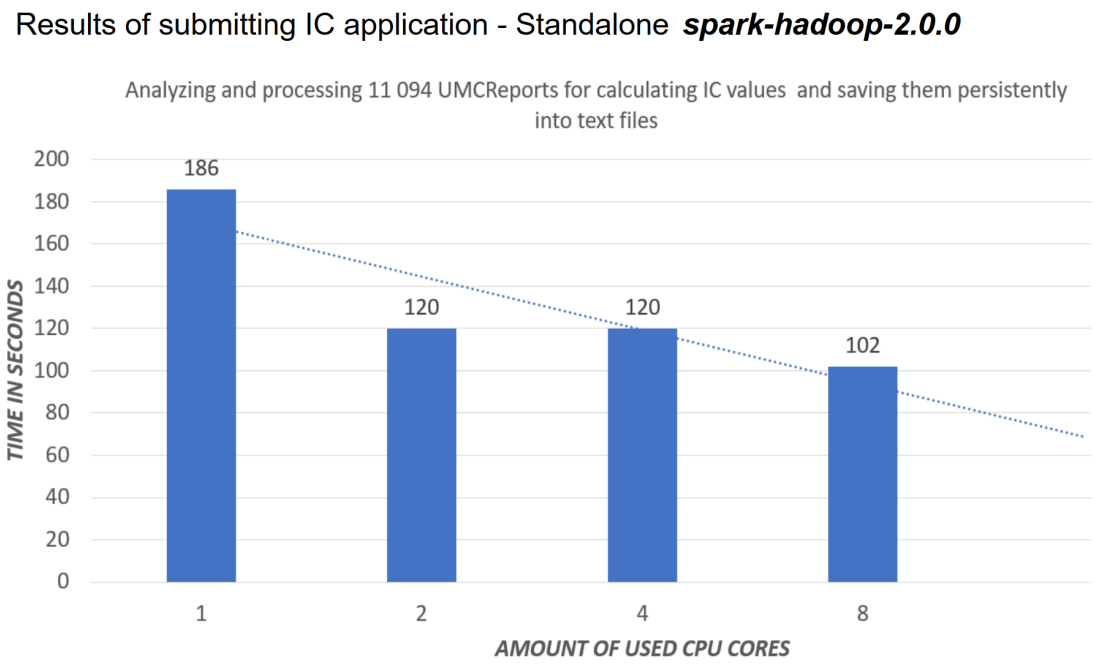

Figure 15: We can see how the runtime is barely decreasing by using more than 2 CPU cores. Since there is a limit to the Input/output Operations Per Second (IOPS) on the hard drive used for saving the IC values, even though increasing an infinite amount of CPU cores, the graph would probably level out after increasing with a certain amount of CPU cores.

The next approach was to try how it could have looked like without any persistent saving operation (with no saveAsTextFile).

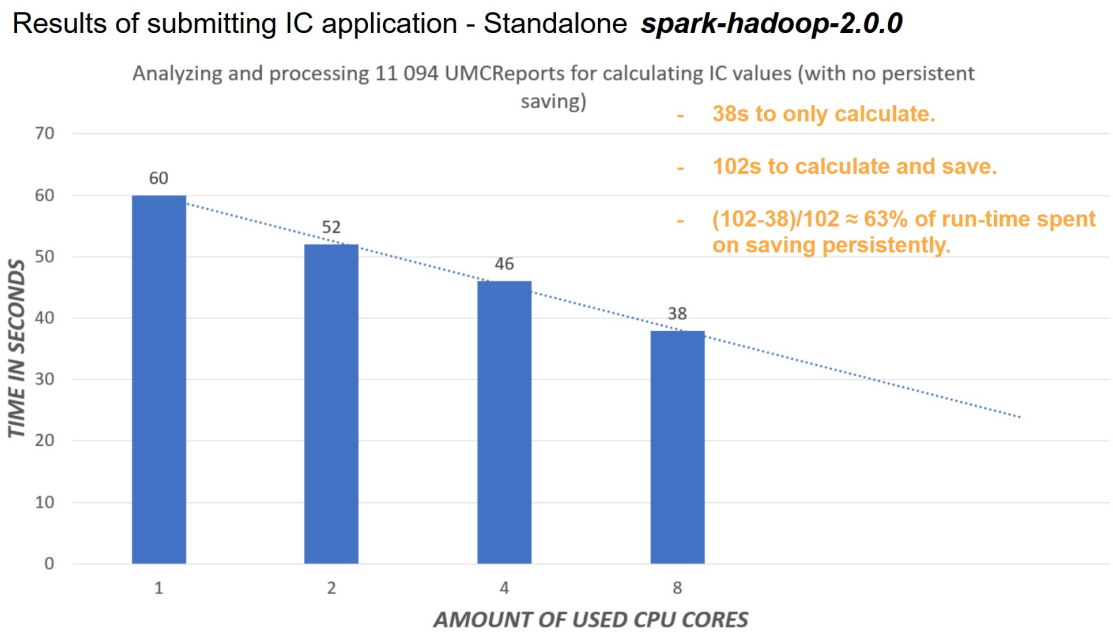

Figure 16: A lot of time were spent on saving the values persistently into text files.

Each host node participating in the Standalone cluster requires Spark installed on it with defined and set node properties. This was accomplished by configuring how the resources on each host node were going to be managed by the executors. The following file had to be edited: */spark/conf/spark-env.sh*. To define the participating host nodes in the Standalone cluster, all the hosts IP addresses has to be inserted into this file on the master host: *spark/conf/slaves*.

Spark on top of YARN - performance measurements and tests

When submitting a Spark application on top of a YARN cluster, it is not required to have Spark installed on each computer node within the cluster. Hadoop is however required to be installed on these nodes since we are actually deploying a Spark container on a HDFS YARN cluster. This was done by configuring four specific XML files plus one shell script file on each computer node. All the properties of the HDFS YARN cluster were defined and specified. Even though a Hadoop cluster were set up, all the UMCReports were read locally from the master node and not loaded from the HDFS storage system which is the whole point of reading and writing data from a distributed data storage.

The UMCReports should have resided distributed across the DataNodes close to its NodeManagers for enabling faster data access during execution. The applications results were actually written back to the Hadoop cluster in a HDFS distribution across the cluster nodes. The result was therefore retrieved in distributed collections of text files across the DataNodes.
It took 3min 51sec for calculating the IC values for a total of 11094 UMCReports when submitting the IC application on the established HDFS YARN cluster.

7 Discussion

7.1 Spark recommendations for submitting applications smoothly

File storage management systems for optimal results

Since Spark is a processing-tool used for managing big datasets, most Spark jobs will likely have to read input data from an external source. File storage management systems are considered as external sources such as the Hadoop Distributed File System (HDFS). In HDFS YARN it is important to place all the Spark workers as close to the HDFS cluster as possible when submitting Spark applications on top of a HDFS cluster. It is important to place the executors close to the data to be analyzed and processed, to decrease time spent on accessing the data before it can be processed. The data to be processed should preferably be placed on the same host node as the executors. It is however optional to place the data collections externally as close as possible to the processing nodes. It would then mean that data has to be loaded in from an external storage into a primary storage, i.e. to the hard drive of the host node processing it. Thereafter loaded into the Random-Access Memory (RAM) of the worker node for execution. With HDFS we directly skip the external part and can start loading into the RAM for execution. The HDFS can be compared with the analogy to a Redundant Array of Independent Disks (RAID) where multiple physical disk drive components are combined into a single logical unit. The RAID presents a storage system on a physical computer machine. This gives the advantage when it comes to Input/output (I/O) operations made against the file storage system. The RAID enables and allows multiple reading and writing operations to be performed at the same time on the logical unit since the operations can all be made on different parts of the RAID and still be operating in an isolated manner on the storage system. HDFS performs just like a RAID but instead of placing all the data collections on a single physical computer machine such as a server, the data collections are distributed across a HDFS cluster. Hard drives from different computers are aggregated across multiple physical machines through Secure Shell (SSH) into a HDFS. Apache Spark recommends the following [9]:

- Run all Spark applications on the same nodes as the HDFS, if possible. This is done by setting up a Spark standalone mode cluster on the same

nodes as the HDFS and configure both Spark's and Hadoop's memory and CPU usage to avoid interference with each other's resources. Alternatively, running Spark and Hadoop on a cluster manager that is common for them both, like Hadoop YARN or Mesos.

- If the previously mentioned point is not possible. Spark should be running on different nodes within the same LAN as the HDFS.
- Data stores that are used for low-latency (like HBase), is preferable to run their computing jobs on diverse nodes than at the same nodes as the storage system is hosted on, this for avoiding interference with each other.

When implementing the IC application, a total data set of 11094 JSON data sets were brought on a USB drive. All the JSON data sets were transferred into a local map on the master host, PC_1. The application actually read all the input (that is the UMCReports) from a local storage and thereafter distributed the data sets through SSH to its executors for execution. This is however not the optimal solution considering the choice of file storage management. A storage management system should be set up and the Spark executors across the cluster should read its distributed data collections respectively and directly from their belonging DataNode in the HDFS. This means that the 11094 UMCReports should be spread across and between each DataNode existing in the Hadoop cluster so that the executors can directly access it when requested for execution, but also write back to it. For instance, to a cloud-platform such as Amazon Web Services or a HDFS. This storage system should also be placed within the same Local Area Network (LAN) as the Spark executors or as close as possible to them for receiving optimal results.

Local Disk

The Spark design is to preserve intermediate output between stages and the Spark nature will try to keep computations in memory as much as possible between job steps for preventing and avoiding to spend unnecessary time on read operations. Spark applications however still has to use local disks to store all the datasets that does not fit in the RAM. This is achieved by Creating a data structure and having a specific partitioning strategy where the data avoids to be shuffled around between each stage of a Spark job, and where it can efficiently be updated without having to serialize and writing everything to disk between each stage. This is why Spark is said to be up to hundred times faster than the present MapReduce oriented technology[9].

Spark and Hadoop will actually perform equally when it comes to writing an application that computes a value by reading it from each input line in the dataset and thereafter writing it back to the disk. Although Spark has a faster start-up time than HDFS, it will not really matter when spending a large amount of time on analyzing and processing data in single steps. While Hadoop writes everything to disk between its execution phases, Spark will fight its in-memory

nature that causes out of memory problems. When Spark cannot hold a RDD in memory anymore, Spark will spill it back to the disk.

Memory

The amount of required RAM strongly depends on how much data that are loaded into the Spark application for analysis. Memory usage required by the applications are greatly affected by the storage level as well as of the serialization format which can be optimized and tuned by configuring the cluster. Serialization is the process of translating RDD into a format that can be stored persistently, such as a text file. It is recommended to allocate at most 75 percent of the RAM of a physical computer for Spark. The other 25 percent is for the Operating System and the buffer cache. Spark can run well with anywhere ranging from 8GB to 200GB of RAM per machine. It is however important to note that JVM processes do not always perform well with more than 200GB of RAM assigned to them. If handling larger amount of memory than this, it is preferred to spread the memory across several workers. Each worker assigned with an even number of CPU cores instead of pushing to much memory into a single worker node that will allocate all the available RAM and all the available CPU cores on the computer. [9]

The amount of RAM used in these thesis experiments were hardly near the limit of the recommendations. The tests were however only going to give the run times of analyzing and processing 11094 fake UMCReports for calculating IC values for the given sub-set which is not a real case scenario for the UMC. The ICSR database is much greater than that.

Network

Many Spark applications are network-bound and it is therefore preferred to use a 10 Gigabit or faster network. This is the best way to make Spark applications run faster on top of multiple node clusters. Multiple nodes on a cluster means that the cluster will have a larger frequency of use of the Secure Shell (SSH) for establishing the cluster, since there exist many computer nodes hosting the cluster. This means that the clusters capacity will strongly be affected by the network capacity as well as it already is affected by the amount of CPU cores present within the cluster but also the amount of available RAM assigned for each executor in the cluster. Clusters are especially network-bound when it comes to Spark transformations that applies ”distributed reduce phases” such as Group-By, Reduce-By or SQL joins. This kind of operations are pipe-lined together through the network into one set of tasks in each stage. The amount of shuffled data involves the redistribution of data across the nodes in the cluster over a network, preferably over a wire such as an Ethernet cable between the worker nodes that are resided on different host nodes. How much the network capacity affects a Spark submit of an application can be monitored in the Spark Web UI. [9]

The setup Standalone cluster in the experiments had a network capacity of 1 Gigabit which is also one factor that affects the run time in total, since it is not the optimal recommended network bandwidth for a Spark cluster to be hosted on.

Central Processing Unit cores

Since Spark performs minimal sharing between the threads managed by the executors, Spark scales well to tens of CPU cores per machine. Each machine in the cluster, i.e the computer nodes within the cluster should preferably have a minimum of 8-16 CPU cores. [9].

That Spark scales well to tens of CPU cores means that Spark scales well when increasing at least about ten CPU cores between each submitting of an application in Spark. To increment with 1,2,4,8 CPU cores as in the experiments in this research and presented in in the results in Section 6, does not really show the effects of how scaleable Spark actually are. Submitting performance tests by increasing the amount of CPU cores in the following order 8,16,32 should probably give the answer to how scaleable Spark actually are. Virtual CPU cores presented by Intel's hyper-threaded processors should never be treated as physical CPU cores.

I/O- or CPU-bound applications

A CPU-bound application means that the rate at which a process finishes, is limited by the speed of the CPU. A task that is performing calculations on a small set of numbers, for instance multiplying small matrices is likely to be a CPU-bound program. A program is said to be CPU-bound when the program would be executed faster if the CPU would be faster. An application is therefore said to be CPU-bound when the application spends the majority of its runtime using the CPU.

An I/O-bound application however means that the rate at which a process is completed is limited by the speed of the I/O subsystem. An application that is dependent on processing data from the hard drive, for instance counting the number of lines in a text file is likely an I/O-bound program. This means that an I/O-bound application spend the majority of its run time doing read from disk and writes to disk [3].

The IC application spent the majority of its runtime writing the IC values to the hard drive, about 60 percent of the runtime were actually spent on writing the IC values to text files. In this thesis when running the IC application, it is for sure I/O-bound since the increment of the amount of CPU cores did not decrease the runtime. This is caused by the storage management system along with the hard drive used in the experiments.

It is however important to note that Spark is a processing-tool that is preferably and directly applied on a distributed storage management system and the whole purpose of developing Spark was to apply Spark as a choice of processing-tool externally on top of a storage management system. The results of these

specific tests are therefore I/O-bound.

Linear scale-ability is impossible to achieve as long as an applications architecture includes making calls over the wire for reaching data for running its processes. Linear scale-ability can only be achieved when the business logic to be executed are resided in-memory together with the data. In that case, the application will only be RAM and CPU dependent, we could just add more hardware for handling our applications throughput demands.

7.2 Standalone Cluster mode

If the Standalone cluster performance tests would only be locally submitted, the delivered test results would reflect a local cluster scenario. The purpose of this thesis was to investigate scalability of the Spark processing-tool as a choice for processing and analyzing ICSRs.

Each and every worker instance (i.e. an executor in Standalone cluster) are spawned as a JVM process in the Standalone cluster. If a cluster is hosted locally it means that there is no shuffling between the computer's executors during runtime. A locally hosted cluster is only one computer host where the established cluster resides. There would only be data transfers locally between the JVM processes on the computer where the cluster is hosted. A locally hosted cluster is therefore said to be only CPU-bound or RAM dependent. Large Standalone clusters means that many computer nodes are linked with each other as well as many worker instances on these computer nodes. This means that a lot of hardware are linked to each other in a clustered environment where the data has to be shuffled over wire between stages, preferably over an Ethernet cable. The shuffling can be prevented if the data collections to be processed are located at the same computer nodes as the executors just like in HDFS. The data would then be read locally by the executors. The executors would also write the results back to the local hard drive. Making a local submit on an eight CPU cored computer would be an unrealistic cluster scenario since the cluster would not be network dependent at all. The node distribution in the cluster has to be considered for providing a realistic established cluster scenario. Eight worker instances were spread over PC_1 and PC_2 with one CPU core for each worker instance and with 5GB of RAM each per worker instance.

As unnecessary runtime are spent on writing the result data to the hard drive (saveAsTextFile in Spark) 6 . This means that the Spark core engine works well when it comes to processing the ICSRs and analyzing them for calculating IC values, but maybe not so well for persistently saving these calculated IC values. As long as the proper hardware and file storage management system are not established for persistently saving the results, significant performance results cannot be expected from Spark.

The implemented IC application is likely to be Input/output (I/O)-bound when running studying the results from Section 6 . The submitted application spent the majority of its runtime doing writes to disk of the IC values after it had finished calculating them. Fortunately, there are different options of cluster

managers in Spark. The cluster manager handling these write to disk difficulties that the Spark core engine comes with, are solved by the HDFS YARN cluster manager. The integration of using Spark as a processing tool and HDFS YARN as an infrastructure for distributing data sets across multiple nodes close to the executors could be an alternative solution and it has the following benefits:

- Distributed data collections should be close to the executors. The benefits that can be expected in this kind of distributed data storage are extended I/O throughput, extended data storage as well as extended processing capability when Spark is attached to it.
- The same pool of cluster resources can be dynamically be shared and centrally configured among all the frameworks that run on the HDFS YARN.
- YARN features can be used such as YARN schedulers for categorizing, isolating and prioritizing workloads in the HDFS.
- Standalone mode requires that each application runs an executor for every host participating in the cluster (all the nodes), while in YARN you can choose the number of executors to use.
- Clusters can use secure authentication between its processes by allowing Spark running against Kerberos-enabled Hadoop clusters. Keberos authentication protocol allows nodes within a cluster to communicate over a non-secure network. This for allowing the nodes in the cluster to prove their identity to one another in a secure manner.

7.3 Spark engine on top of HDFS YARN

The expected results from submitting the IC application on top of the Hadoop Distributed File System (HDFS) Yet Another Resource Negotiator (YARN) cluster were not as expected. When submitting the IC application on top of the Standalone cluster and including saving the results persistently to disk, most of the run time were actually spent on writing the IC values do disk and not on computing the actual IC values. Since the data collections to be processed (the Individual Case Safety Reports, ICSR's) were not resided on the executor nodes, a lot of run-time were spent on shuffling the data to its executors through Secure Shell (SSH) which is the contrary to the purpose of the HDFS YARN architecture. Distributed data collections should be resided close to the executors, preferably at the same nodes. Improvements in run time were actually expected but unluckily they were not seen. In Standalone cluster mode:

- It took 38 seconds to calculate all the IC values for 11094 UMCReports.
- It took 102 seconds to calculate all the IC values but also saving the IC values persistently onto disk for the same amount of UMCreports.

- This means that 102-38 = 64 seconds were spent on writing the text files containing the IC values onto disk. That is an average of 64/102 = 63 percent of the total runtime that were spent on writing the IC values.

This analysis shows that there were persistent saving performance issues when submitting the IC application on top of the HDFS YARN cluster. Since there are many HDFS YARN properties to adjust and set for utilizing every resource out of the cluster, optimal performance results cannot be expected.

Considering that the UMCs current SQL server has limited Input Output Operations per Second (IOPS) that can be made against their database, HDFS might be a better choice of data storage management instead of placing all the data on a single server which is the old-fashioned way of managing data.

Input/Output (I/O) issues are usually improved by providing another option of file storage management but also by selecting the right hardware that both will give reduced run times. This can be accomplished by reading all the input data from a file storage management system such as a HDFS. Not only reading the data from this storage management system but also writing the results back to it when all the IC values are calculated. The run-times can be improved by satisfying the cluster with the right hardware.

Having all the available resources provided in a YARN cluster does not mean that all the executors will utilize every bit of resource if not properly configured and tuned. There are various configurations and various factors that has to be counted in and that affects the work flow when submitting an application on top of a HDFS YARN cluster. These are the following factors that all YARN clusters performance depends on [7]:

- Number of used executors for each spark job.
- Memory allocation.
- Container allocation.
- Virtual core allocation.
- Scheduling policy.
- Queuing policy.

Since none of the properties mentioned above were considered when setting up the HDFS YARN cluster in this experiment, this has to be taken under consideration when analyzing the results. HDFS YARN should not be judged by the failed performance measurements brought in this thesis. As long as an ETL-architecture are dependent on I/O or network overhead, linear scalability will never be seen since these are the cons of all processes.

7.4 Scalability of Spark for processing ICSRs

In enterprise systems, network overhead are the destroyer of scalability since runtime has to be spent on communicating between processes and not on the

actual execution. If the primary goal is to achieve linear scalability, avoiding shuffling between processing nodes has to be considered when building ETL-flows.

1. The router that hosted the network used by the cluster had 1 Gigabit Ethernet WAN ports. This is 10x less than the minimum recommended network capacity.

2. All the JSON data sets were read from the local disk drive and all the calculated result values were written back to it. This is not the preferable way to manage data when using Spark. File storage systems are usually used for the purpose of faster access. Preferably, the data should be resided in-memory for enabling fastest possible access for the executed process.

3. Spark recommends memory from 8GB of RAM up to 200GB. The nodes in this experiment had 5GB. More memory provided, means less write to disk operations between each stage of execution which means decreased run-times, since the intermediate results do not have to be written to the disk between each step or phase of a process executing. Spark allocates the RAM up to its threshold limit, it tries to allocate the default in-memory for as many tasks as possible to sure that the in-memory are totally utilized. When Spark runs out of workable memory for a new job, Spark will try to spill the in-memory content of least recently used Resilient Distributed Datasets (RDDs) to disk so that Spark can allocate new jobs for execution. For a huge amount of Spark jobs running on a low in-memory machine, then most of the RDDs will be placed in the disk only since the in-memory has not enough room for them all. This means that all the Spark applications running on an average consumer computer will disable the advantage of Spark in-memory processing. The application will therefore spend unnecessary time on reading and writing between its processing steps just like the behavior of HDFS YARN.

4. File storage management systems are usually hosted on computer machines with server hardware that are quite different in performance characteristics from the average consumer disk that was used in the experiments. A group of solid-state drives (SSD's) are usually connected together and aggregated to create a Redundant Array of Independent Disks (RAID) for these kinds of purposes. The Operating System (OS) considers these connected disks as one large disk just like a HDFS cluster is considered as one single file storage management system. By stacking several SSD's to one large RAID enables read and write operations to be spread out over multiple disks which means that the Inputs/Outputs (I/O: s) can be carried out simultaneously instead of e.g. queuing the IC values that are going to be saved into text files.

Therefore, no matter how a cluster scales with its workloads, the optimal utilization that Spark is said to come with will not be reflected. After scaling

out the workload to a certain point for a large amount of worker nodes, the IC application will probably not finish faster and the graph would probably be totally flatten out. At this point, it is no longer a manner of the amount of CPU cores or RAM that affects the run time but more likely depending on the data to read as input and in what way it is served for the Spark executors. Because of the in-memory nature of most Spark computations, Spark applications can be bottle-necked by any resource in the cluster, that is the CPU, RAM or network bandwidth. All these resources have to be considered before attempting to run applications in Spark. It is not certain even if a cluster has the perfect environment, that a linear decrease of the run time will be seen when increasing the amount of CPU cores. An improvement should be expected but most certainly not guaranteeing an even linear reduction. If the data to be processed are resided in-memory within the ETL-process, the minimal required resources for executing this process would be RAM and CPU. Adding more hardware would mean handling more throughput.

1. Appropriate file storage management system configured for the demands required.

2. Run the storage management system on appropriate hardware. Preferably not a cluster of laptops.

3. Consider the Spark application, what precautions can be made and what part of the application is really necessary to avoid unnecessary executions. Spark features performing the same thing but in different ways should be tested and compared.

8 Conclusions

All the unnecessary intermediate read and write operations can be avoided in Spark by utilizing the Random-Access Memory which is not the case in the current solution. Since a database transaction usually contains a sequence of SQL commands, the result for the SQL command's respectively has to be written back to disk and reached from disk. This Input/output (I/O) issue is solved in Spark since Spark utilizes the RAM before writing the data to disk. Another strong argument for using Spark is that everything is executed in parallel, which is not the case for the current processing system at the Uppsala Monitoring Centre. Diverse partitions are assigned to diverse executors in Spark which ensures that the executors will never interfere with each other's memory spaces. This gives both extended execution capability as well as extended I/O capability and enables a distributed work flow. All the individual case safety report (ICSR) data in VigiBase has to be reached and executed concurrently in isolated manners by a single SQL server which means that workloads are queued. This means that the UMC's member countries have to wait some additional days, weeks or months before they can retrieve the Information Component (IC) values. This is not the optimal way to treat real-time data.

For a well established cluster environment, Spark is a great processing tool for real-time data analytics and processing. Fresh data values such as IC values that are strongly dependent of the real-time data information of all the existing drug-Adverse drug reactions (drug-ADrs) experienced out in the world, requires to be re-calculated almost instantly as a single ICSR arrives for delivering real-time results.

The investigated experiments in this thesis show that it is possible to reduce the run time of calculating IC values by scaling out the workloads on multiple computer nodes. Maybe not as drastically as expected but there are many factors that has to be considered when studying the delivered results. Since none of the cluster environment were considered at all before performing the performance measurements and tests, an expectation of linear reduction of the run time is a dream case.

9 Future work

To re-establish the entire VigiBase into a HDFS where all the ICSRs are resided on multiple nodes may be way too complex. Another option for the UMC is to integrate Spark streaming with their current batch-processing system or possibly the cloud as a choice of external storage management system. The data to be analyzed and processed has to be reached and scheduled for execution from a storage management system. If placing safety data up to the cloud in the purpose of analysis and process, it will lead to another security issue that also has to be managed somehow since all the ICSRs has to be protected.

The data has to be close to the executors for fast access so that the long-term scheduler can load the data into the Random-Access Memory. In Spark, RDDs are converted and retrieved from a DataFrame when database transactions are made against the DataFrame. All the ICSRs should therefore arrive in a stream and fill up a Spark DataFrame each and every day. This Spark DataFrame would contain all the ICSRs existing in VigiBase plus the ones arriving since it is integrated with a batch-processing stream. Since a single ICSR affects all the other IC values, it would be unwise to recalculate all the IC values just as an ICSR arrives. The UMC should instead set a 24h loop on this DataFrame and execute all the IC values once a day. In that way, they can guarantee their member countries fresh IC values each and every day.

References

[1] VIPS - Architectural overview, UMC, (2016). [UMC documents] Not available publicly.

[2] IC - Disproportional reporting, UMC, (2016). [UMC documents] Not available publicly.

[3] Operating system concepts 9th Edition by Hoboken, Silberschatz, Abraham; Galvin, Peter Baer.; Gagne, Greg. ,2013 Publisher: N.J Wiley

[4] Computer Architecture: A Quantitative Approach 5th Edition by John L, Silberschatz, Abraham; Galvin, Peter Baer.; Gagne, Greg. . Hennessy and David A, 2011

[5] Learning Spark Matei Zaharia, Patrick Wendell, Andy Konwinski, Holden Karau , 2015 Publisher: O'Reilly Media, Inc. ISBN: 9781449359034

[6] Learning SQL, 2005 Publisher: O'Reilly Media, Inc. ISBN:978-0-596-00727-0

[7] Rishi Yadav, Spark Cookbook, Publisher: Packt Publishing, 2015 (2016).

[8] Tom White, Hadoop: The Definitive Guide, 4th Edition Storage and Analysis at Internet Scale, Publisher: O'Reilly Media (2015).

Webb references

[9] Apache Spark, (2016). [online] Available at: $http://spark.apache.org/$ [Accessed 16 July.2016]

[10] Description of ICSR: Basic Facts, FDA, US Food and drug administration, (2016). [online] Available at: $http://www.fda.gov/ForIndustry/DataStandards/IndividualCaseSafetyReports/$ [Accessed 17 November.2016]

[11] Database System: Basic Facts, VigiBase, the WHO Global ICSR, (2016). [online] Available at: $http://www.who-umc.org/graphics/24965.pdf$ [Accessed 17 November.2016]