Scaling out Big Data Distributed Pricing in Gaming Industry

Eero Soikkeli

School of Science

Espoo 07.01.2019

Supervisor

Prof. Petri Vuorimaa

Advisor

M.Sc. Antti Hätinen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Author Eero Soikkeli

Title Scaling out Big Data Distributed Pricing in Gaming Industry

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Petri Vuorimaa

Advisor M.Sc. Antti Hätinen

Date 07.01.2019 **Number of pages** 60 **Language** English

Abstract

Game companies have millions of customers, billions of transactions and petabytes of other data related to game events. The vast volume and complexity of this data make it practically impossible to process and analyze it using traditional relational database models (RDBMs). This kind of data can be identified as Big Data, and in order to handle it in efficient manner, multiple issues have to be taken into account. It is more straightforward to answer to these problems when developing completely new system, that can be implemented with all the new techniques and platforms to support big data handling. However, if it is needed to modify an existing system to accommodate data volumes of big data, there are more issues to be taken into account.

This thesis starts with the clarification of the definition 'big data'. Scalability and parallelism are key factors for handling big data, thus they will be explained and some of the conventions to do them will be reviewed. Next, different tools and platforms that do parallel programming, are presented. The relevance of big data in gaming industry is briefly explained, as well as the different monetization models that games have. Furthermore, price elasticity of demand is explained to give better understanding of a Dynamic Pricing Engine and what does it do.

In this thesis, I solve a bottleneck that emerges in data transfer and processing when introducing big data to an existing system, a Dynamic Pricing Engine, by using parallel programming in order to scale the system. Spark will be used to deal with fetching and processing distributed data. The main focus is in the impact of using parallel programming in comparison to the current solution, which is done with PHP and MySQL. Furthermore, Spark implementations are done against different data storage solutions, such as MySQL, Hadoop and HDFS, and their performance is also compared.

The results for utilizing Spark for the implementation show significant improvement in performance time for processing the data. However, the importance of choosing the right data storage for fetching the data can't be understated, as the speed for fetching the data can widely variate.

Keywords Big Data, Scalability, Parallelism, Parallel Programming, Parallel Processing, Gaming Industry, Monetization, Distributed Computing, Spark, Price Elasticity of Demand

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi

Tekijä Eero Soikkeli

Työn nimi Scaling out Big Data Distributed Pricing in Gaming Industry

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Prof. Petri Vuorimaa

Työn ohjaaja M.Sc. Antti Hätinen

Päivämäärä 07.01.2019 **Sivumäärä** 60 **Kieli** Englanti

Tiivistelmä

Peliyhtiöillä on miljoonia asiakkaita, miljardeja maksutapahtumia ja petatavuja pelin tapahtumiin liittyvää dataa. Tämän datan suuri määrä ja kompleksisuus tekevät sen prosessoimisesta sekä analysoimisesta lähes mahdotonta tavallisilla relaatiotietokannoilla. Tällaista dataa voidaan kutsua Big Dataksi, ja jotta sen käsittely olisi tehokasta, useita asioita on otettava huomioon. Uuden järjestelmän toteutuksessa näihin ongelmiin pystytään vastaamaan melko johdonmukaisesti, sillä uusimmat tekniikat ja alustat voidaan ottaa tällöin helposti käyttöön. Jos kyseessä on jo olemassa oleva järjestelmä, jota halutaan muuttaa vastaamaan big datamaisiin datamääriin, huomioon otettavien asioden määrä kasvaa.

Tämän diplomityön aluksi selitetään termi 'Big Data'. Big Datan kanssa työskentelyyn tarvitaan skaalautuvuutta ja rinnakkaisuutta, joten nämä termit, sekä näiden yleisimmät käytännöt käydään läpi. Seuraavaksi esitellään työkaluja ja alustoja, joilla on mahdollista tehdä rinnakkaisohjelmointia. Big Datan merkitys peliteollisuudessa selitetään lyhyesti, kuten myös eri monetisaatiomallit, joita peliyritykset käyttävät. Lisäksi kysynnän hintajousto käydään läpi, jotta lukijalle olisi helpompaa ymmärtää, mikä seuraavaksi esitelty Apprien on ja mihin sitä käytetään.

Tässä diplomityössä etsin ratkaisua Big Datan siirrossa ja prosessoinnissa ilmenevään ongelmaan jo olemassa olevalle järjestelmälle, Apprienille. Tämä pullonkaula ratkaistaan käyttämällä rinnakkaisohjelmointia Sparkin avulla. Pääasiallinen painopiste on selvittää rinnakkaisohjelmoinnilla saavutettu hyöty verrattuna nykyiseen ratkaisuun, joka on toteutettu PHP:llä ja MySQL:llä. Tämän lisäksi, Spark toteusta hyödynnetään eri datan säilytysmalleilla (MySQL, Hadoop+HDFS), ja niiden suorityskykyä vertaillaan.

Tulokset, jotka saatiin Spark toteutusta hyödyntämällä, osoittavat merkittävän parannuksen suoritusajassa datan prosessoimisessa. Oikean tietomallin valitsemisen tärkeyttä ei pidä aliarvioida, sillä datan siirtämiseen käytetty aika vaihtelee myös huomattavasti alustasta riippuen.

Avainsanat Kysynnän hintajousto, big data, pilvilaskenta, rinnakkaisohjelmointi

Preface

I want to thank Professor Petri Vuorimaa and my instructor M.Sc. Antti Hätinen for their good guidance. I would also like to thank PHZ Full Stack Ltd. and Apprien Ltd. for providing me the possibility to make the thesis from this topic. Lastly, I want to thank my family and friends for their constant support throughout the entire process.

Otaniemi, 07.01.2019

Eero J. Soikkeli

Contents

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
EMR	Elastic Map Reduce
ETL	Extract, transform, load
FLOPS	Floating point operations per second
GPGPU	General-purpose graphics processing unit
HDFS	Hadoop Distributed File System
HPC	High-Performance Computing
IAP	In-App Purchase
JVM	Java Virtual Machine
MPI	Message Parsing Interface
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
SDK	Software Development Kit
SGD	Stochastic Gradient Descent
YARN	Yet Another Resource Negotiator

1 Introduction

The amount of data is steadily rising, and the hardware and software solutions, of both new and old systems, need to be able to answer that. The amount and complexity of data is so vast, that it can't be processed by traditional methods. This kind of data is known as Big Data. Methods to process Big Data revolve around distributed systems and parallel computing. At hardware level, multicore processors are utilized to do the processing simultaneously between all cores. At software level, technologies and platforms have been developed, that makes it simple to implement programs that can be executed in parallel.

Data analysis is valuable for all industries, including game industry. Nowadays game companies collect petabytes of data, that could be used to improve multiple aspects in the games, assuming the data is used correctly. Using the data to plan monetization is one of these options. Prices can be set, with correct analysis, in a way that the product price is in par with its value to the players. One way to achieve this is by using price elasticity of demand together with regression analysis.

In this thesis, the main goal is to scale an existing application to handle big data workloads. More accurately, the research question is how to scale an existing application to handle vast data volumes The goal of the research is to find suitable tools, that enable parallelism and are practical to use to improve the overall efficiency of the system. The main focus is on the differences in time complexity for the different implementations.

The structure of this thesis after this Introduction chapter is as follows: First, the background for big data, scaling and parallelism is reviewed, and the basic methods for scaling and parallelism are explained. Next, the tools and platforms that are popular, convenient and interesting for resolving big data problems, are looked into. After that, the role of big data and the different monetization models in gaming industry are inspected. In this chapter price elasticity of demand is also explained in more detail and different formulas for solving it are presented. Furthermore, the application that is being scaled, a Dynamic Pricing Engine, is introduced with the explanation of its default dataflow. The next chapter, Research Material Methods consists of the reasoning for the tools and methods that are utilized to answer the scalability issue. In addition, this chapter consists of implementation prototypes, whose performance will be compared both to the original implementation and to each others. After this, the results gained by using the previously mentioned methods are presented alongside the comparisons of each method. Lastly the whole thesis is reviewed in the Summary chapter.

2 Background

2.1 Big Data

The definition of big data is somewhat difficult to get right. One definition of big data is that it "represents the progress of human cognitive processes, usually includes data sets with sizes beyond the ability of current technology, method and theory to capture, manage, and process the data within a tolerable elapsed time" [23]. Another definition is that "Big Data are high-volume, high-velocity, and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization" [23]. Lastly Wikipedia description goes as such: "In information technology, big data is a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools"[23]. Thus, the large volume of data is not the only property of Big data. Another properties are velocity, variety and veracity. Velocity refers to the amount of data which is growing at high speed. The multitude of data formats refer to the variety. Veracity can be thought as the uncertainty that is involved in knowing if data is right or not [47]. Big Data is not a new concept, but it is very challenging. The fundamental fact is that data is too big to process conventionally.

Managing big data is one of the most important challenges in modern systems. Due to its volume, complexity, and speed of accumulation, it is hard to manage and analyze Big Data manually or by using traditional data processing and management techniques [10]. It is also argued that commercial DBMSs (Database Management Systems) are not suitable for processing extremely large scale data [23]. Classic architecture's potential bottleneck is the database server, while faced with peak workloads. One database system server has restriction of cost and scalability, which are two important goals of big data processing [23].

Three important issues that are encountered with big data processing, are Big Data Storage and Management, Big Data Computation and Analysis and Big Data Security [23]. Classic data management systems are not able to satisfy the needs of big data. Furthermore, the amount of data is growing at a much faster rate than storage capacity. Previous computer algorithms are not able to effectively storage data that is heterogeneous, however they perform excellent in processing homogeneous data. Therefore, re-organizing data is one big challenge in big data management.

On the other hand processing a query in big data, speed is in significant demand, but the process may take time, because mostly it cannot traverse all the related data in the whole database in a short time [23]. The combination of appropriate index for big data and up-to-date preprocessing technology will be a desirable solution, when these kind of problems are encountered. Application parallelization and divide-and-conquer is natural computational paradigms for approaching big data problems. If there is enough data parallelism in the application, users can take advantage of, for

example, cloud provider's reduced cost model to use hundreds of computers for a short time costs.

Security and privacy affect the entire big data storage and processing, since there is a massive use of third-party services and infrastructures that are used to host important data or to perform critical operations. Unlike traditional security methods, the main consideration of big data security is how to process data mining without exposing sensitive information of users [23].

2.2 System Scaling

Scalability is defined as the system's ability to handle growing amount of work load in a capable manner or its ability to be enlarged to accommodate that growth [12]. When scaling a system to be able to handle larger amount of data, it is required to think of the ways to achieve this. Typically, system can be scaled up or scaled out also known as vertical scaling and horizontal scaling [40].

Scaling up (vertical scaling) means that when the need of more processing power arises, it is handled by adding more efficient hardware to the processing CPU. This can be more memory, more efficient hardware or faster connections. Scaling up usually involves single instance of an operating system.

Scaling out (horizontal scaling) mean that when the need of more processing power arises, more inexpensive CPUs are added to deal with the problem. The workload is therefore distributed between many servers. Multiple independent machines are grouped together in order to increase processing capability. Typically multiple instances of the operating systems are running on separate machines [40].

Both approaches have advantages and disadvantages. Scaling up is easier to manage, as adding hardware to a single machine is rather straightforward. Furthermore most software can easily take advantage of the addition of more capable hardware. However, scaling up is much more expensive than scaling out, and it is not possible to scale up after a certain limit. Also, the system needs to be powerful enough to handle future workloads, resulting in wasted processing power initially. Scaling out on the other hand is relatively cheap and the performance can be increased in small steps when needed [40]. Also, system can be scaled out as many times at is needed, thus there is no limit how much more processing power can be achieved. One downside of scaling out is the difficulty of its implementation as the system has to support handling distributed data and parallel processing.

Scaling can naturally be done both in local environments and on cloud platforms. On cloud platforms, such as AWS and Azure the scaling doesn't usually require as much as effort as locally. Generally just the limits of how much the system can scale are defined, and the platform adds computation instances depending on the demand. These instances are billed based on the computation time. However, as

mentioned before, scaling locally is relatively easy and it should be used in certain cases, where there are many computations and the workload can be predicted. In these cases, it's typically cheaper to use own hardware and scale it either up or out.

It is important to know the limits of hardware development when thinking about scaling up. Moore's law states that the number of transistors in a circuit doubles approximately every two years [28]. However, in the early 2000s the chips began to heat too much due to the smaller silicon circuits and faster electron movements [46]. The maximum clock rate hasn't improved since 2004, but the amount of transistors per chip has followed Moore's law (See Figure 1). This mean that the processing power of a single processor core has not improved, but the overall processing power has. However, the main issue in utilizing the processing power of these multicore processors, has been the difficulty of parallel programming

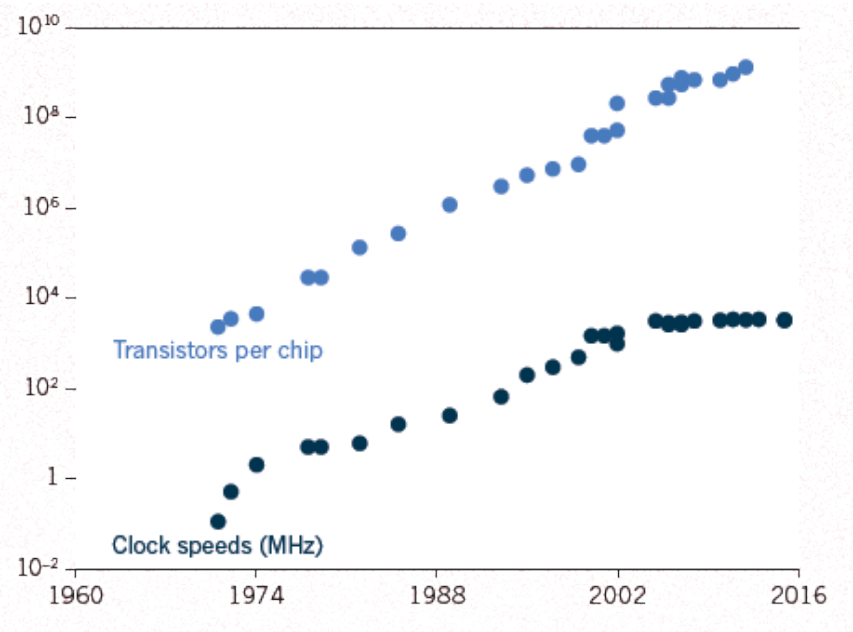

Figure 1: Transistors per microprocessor chip and clock speeds from 1960 to 2016 [46]

Examples of frameworks that support scaling out are peer-to-peer networks, Apache Hadoop, MapReduce and Apache Spark. Platforms that support scaling up are for example Multicore CPUs, High Performance Computing (HPC) Clusters and GPUs (Graphic Processing Units)[40].

2.3 Parallel Programming

2.3.1 Parallelism

There are typically two main instances of parallelism involved in computational tasks. The first type is task-based parallelism, which is usually exploited by a typical operat-

ing system [15]. For example, a user can be writing a Thesis in LateX while playing music from YouTube in the background. Basically, CPU cores can be exploited by running each application on different core. The most common convention to do multiprocessing is context switching, where, for example, CPU processing time is divided into 10ms (100Hz) timeslots, that are allocated to prioritized processes. With multiple CPUs there are more than one servers, but the multiprocessing is still based on context switching. Prioritizing algorithm can also divide the workload between multiple CPUs to even the energy consumption, in order to prevent overheating of the components. To tie this to parallel programming, a program can be written as a number of sections that send the information from one application to another. This pattern in turn is called pipeline parallelism, where the output on one program provides the input for the next [15].

The other type of parallelism is data-based parallelism. This is needed, because while there are GPUs capable of calculating teras of FLOPs, transferring that amount of data will be the bottleneck for these systems. In other words, the increase of memory bandwidth is not keeping pace with the increase of computing power. Network performance is also lacking behind CPU performance [19]. The main idea is to find out how the data can be transformed instead of concentrating on tasks [15].

To compare the two, task-based parallelism fits more with coarse-grained parallelism approaches. This can be deduced by the fact that there is a number of powerful processors that can perform a significant portion of the work. For example, if there would be a number of same size, separate arrays and the task would be to transform them, then on task-based parallelism approach the arrays would be assigned on different CPU cores or GPU SMs (streaming multiprocessors). On the other hand with data-based decomposition first array would be split into blocks and each block would be assigned to one CPU core or one GPU SM. The procedure would then be repeated for the remaining arrays [15].

2.3.2 Parallel Processing

Parallel processing is essential to perform a massive volume of data in a timely manner. The use of parallelization techniques and algorithms is the key to achieve better scalability and performance for processing big data. There are a lot of parallel processing models, including General Purpose GPU (GPGPU), MapReduce and Apache Spark[23].

One crucial theory for parallel processing is Amdahl's law. Amdahl's quantity can be represented with the formula

$$1/(1 - P + P/N) \tag{1}$$

Where P is the fraction of the program that can be parallelized and N the number of processors. From the equation we can see that the bottleneck is in the fraction of irreducibly serial computations in the code, which means that if P is 90%,

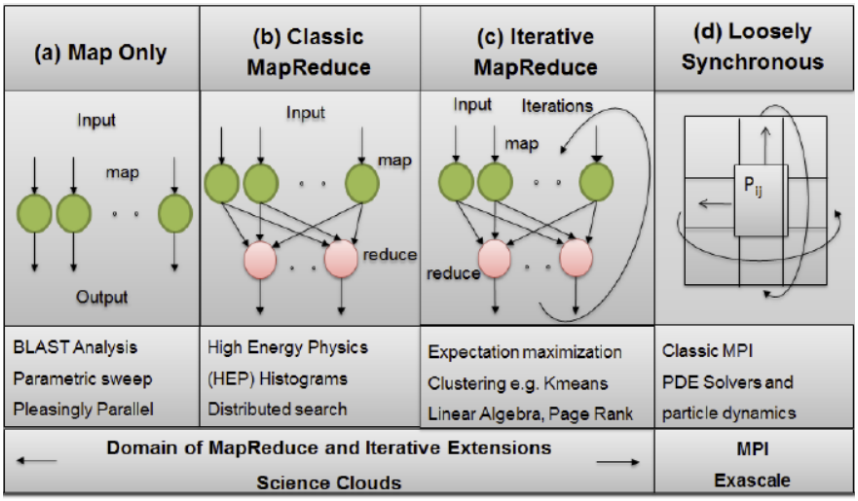

Figure 2: Forms of Parallelism and their applications on Clouds and Supercomputers [14]

the maximum speed-up in only 10-fold even with an infinite number of processors [42].

Programs that are run on massively parallel supercomputers follow strict programming models that are designed to deliver optimal performance and scalability. These programs are designed using a loosely synchronous or bulk synchronous model and the Message Passing Interface (MPI) communication library [14]. The characteristic of this model is that each processor does a little computing and then stops to exchange data with other processors in very precise manner and then the cycle repeats. If the complexity of the communication pattern does not grow as the problem size increases, then this kind of computation scales very well to large number of processors.

MapReduce model is usually utilized in parallel programming, and it can be divided to two variants. In one case, there is no real reduce step, meaning that the operation is just applied to each input in parallel. The other extreme are computations that use MapReduce inside an iterative loop such as machine learning computations [14]. In iterative MapReduce the program iterates over the dataset until predefined requirements, also known as stopping conditions, for the solution are met.

The first mentioned algorithms that are called embarrassingly parallel (or naturally parallel), are the simplest type of parallel algorithms, which require almost no communication between the processes. Each process can perform their own computations without any need of communications with the others. A problem or a task on the other hand is called embarrassingly parallel, when they require little to none effort to separate the problem into a number of parallel tasks. The term 'embarrassingly parallel' has some unnecessary undertones in it. Because if a formula is constructed in a way that the output data points can be represented without relation to each other then the formula is well formed and it can be implemented extremely well on

GPUs [15]. Furthermore, even if one step of the algorithm cannot be presented as embarrassingly parallel step, it is still beneficial, as that stage may turn out to be a bottleneck, but the rest of the problem is relatively easy to implement on a GPU. Only when the problem is constructed in a way, that it requires every data point to know the value of its surrounding neighbors, the speedup will be limited in the end. Adding more processors to deal with the problem works to a certain point, but after that certain point the computation will be slowed down due to the processors spending more time sharing the data than processing it [15].

In this thesis, I will scale the Dynamic Pricing Engine, which uses linear regression. Therefore, it is important to figure out, if it is possible to parallelize linear regression. In Apache Spark, parallelization of linear regression is done by using stochastic gradient descent (SGD) [26]. SGD is used to approximate the gradient using a subset of data, for example, 1 billion items out of 100 billion items. First, the average of each subset is calculated with separate computers. Then, the average of these results is calculated [26]. Here the first part is the map step, and the second part is the reduce step.

2.4 Message Parsing Interface

One programming model that can be used for programming parallel algorithms is MPI (Message Parsing Interface). MPI has zero tolerance for hardware or network errors, but it offers a flexible framework [30]. MPI is namely an Application Programming Interface (API) that defines properly the syntax and software library that provides standardized basic routines to build complex programs. The MPI API does not depend on the underlying programming language it uses. The MPI allows the coding of parallel programs, that exchange data by sending and receiving messages encapsulating that data [30]. One major drawback of MPI is the missing support for fault tolerance, since it mainly addresses HPC problems [36]. Another downside is that is not suitable for small grain level of parallelism, such as using the parallelism of multi-core platforms for shared memory multiprocessing [36].

In his book Nielsen arguments that MPI interface is the dominant programming interface for parallel algorithms with distributed memory in the HPC community [30]. He reasons this with the standardization of many global routines of communications and many primitives to perform global calculations in the MPI.

2.5 MapReduce

MapReduce, which was originally proposed by Google [16], is the basic data processing scheme that breaks tasks into two components, known as Mappers and Reducers, which is based on the master-worker pattern. Master-worker pattern is composed of a master and a number of independent workers. Master assigns tasks to workers, which then execute the assigned tasks independently in parallel [47]. Generally in MapReduce mappers read the data from a filesystem, for example, from Hadoop

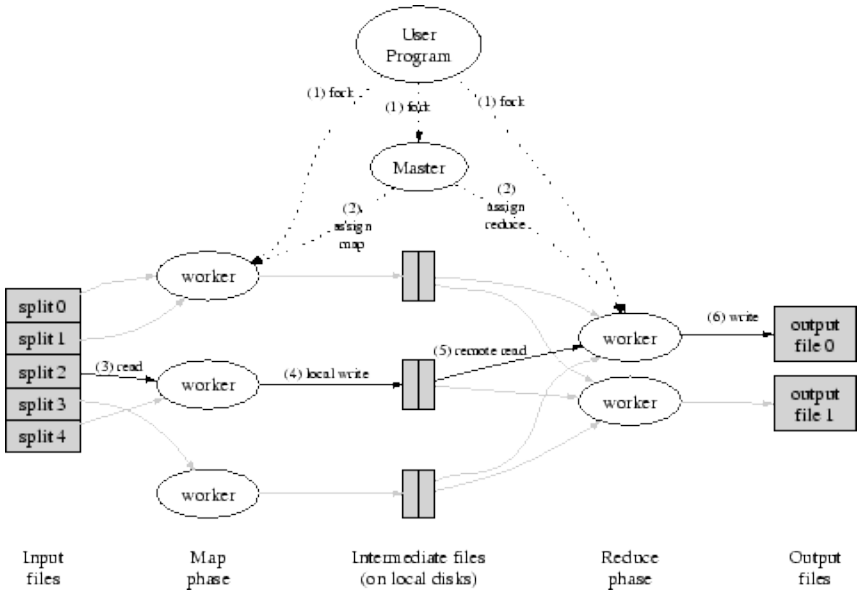

Figure 3: MapReduce Execution data flow [16]

Distributed File System (HDFS), and compute intermediate results to be passed to the reducers. Reducers aggregate these results and generate the final output which is then written to filesystem.

Historically, functional languages, such as Scala, have been slower than traditional imperative languages, such as PHP. Thus, majority of programs have been written in imperative languages, due to the more efficient utilization of the lower processing power in the past [33]. However, processing power of modern computers exceeds the capabilities of imperative programming model. Therefore, although functional languages are generally slower, the gap between them and imperative languages is narrowing, because imperative languages are not able to utilize all of the processing power available.

In functional programming, it is assumed, that when data structure is updated, both the old and the new versions will be available for further processing [33], which means the elements are immutable. In comparison, in imperative languages the old version of the data structure is no longer available. Immutable elements are key feature of functional programming and they are crucial for systems utilizing parallel programming, because, if the data changes throughout the run, it limits the parallelism to a single process.

All data elements in MapReduce are immutable, meaning that they cannot be updated [48]. Changed input (key, value) in a mapping task does not get reflected back in the input files. Only by generating new output (key, value) does the communication occur.

While MapReduce is a viable tool for processing large datasets, it falls short in

iterative processes. Mappers read the same data from disk for each iteration. This results the system to wait the data to be written on disk after each iteration which in turn makes the disk access a major bottleneck [40]. Apache Hadoop and Spark are popular platforms that can be used for effortless MapReduce implementation.

In their study Reyes-Ortiz et.al. [36], compared MPI and Spark with two machine learning algorithms: K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). The result was that MPI outperformed Spark by more than one order of magnitude in terms of processing speed and provides more consistent performance. However, they argue that Spark shows better data management infrastructure and fault management [36].

2.6 Lambda Architecture

One method to handle massive data volumes in online processing environment is using lambda architecture. The basic architecture consists of three layers: A batch processing layer for precomputing large amounts of data sets, a speed computing layer for minimizing latency by doing real-time calculations as the data arrives and a layer to respond to queries and provide the results of the calculations [24]. Lambda architecture allows users to optimize the costs of data processing by understanding, which parts of the data need online or batch processing. Furthermore, the architecture partitions datasets to allow calculation scripts to be executed on them. However, it is argued that the amount of projects that need to be maintained to allow multiple data executions requires more skills from the developers setting up the jobs to produce results [24].

In short, the lambda architecture is well suited for big data processing. The online stream can be used to detect data anomalies verifying, whether it is accurate before processing it further. This verified data can then be stored into databases, which can have batch scripts performed once a day or a week to study data patterns over a time period [24].

2.7 GPU Programming & Computing

Graphics processing units were originally designed to output a 2D image from 3D virtual word, which mean that the operations it could perform where fundamentally linked with graphics, as the name suggests. Using GPUs for non-graphics applications has been a recent point of interest, and the use of GPUs has now matured, leading to better support from programming languages and tools for utilizing GPU computing. This computation is commonly known as GPGPU [13].

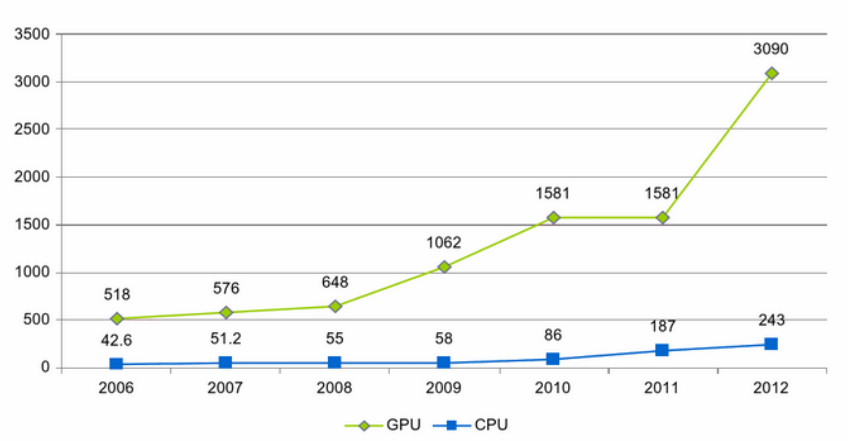

Figure 4: GPU and CPU peak performances in gigaflops from 2006 to 2012 [15]

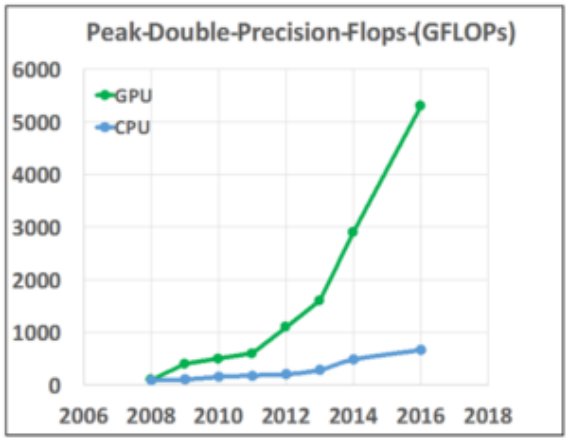

Figure 5: GPU and CPU peak double precision in gigaflops from 2008 to 2016 [29]

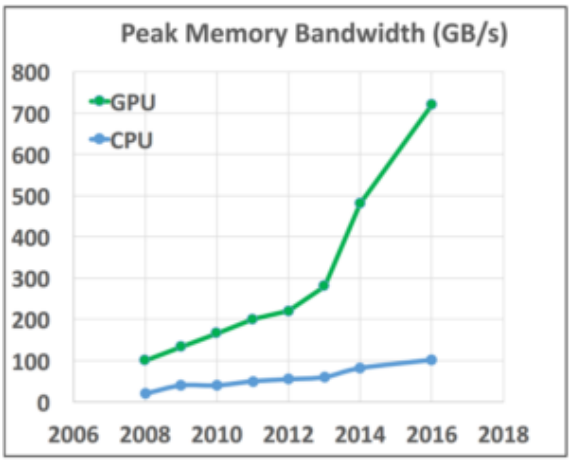

Figure 6: GPU and CPU peak memory bandwidth from 2008 to 2016 [15]

Major aspect of why GPU computing has been successful, is its massive performance compared to the CPU. The performance gap can be about seven times between the two when comparing theoretical peak bandwidth and gigaflops performance [13]. FLOPS is an acronym for computing term floating point operations per second and it measures computer performance. As CPU is basically a serial von Neumann processor and optimizes the execution a series of operations in order, the performance could be increased by increasing frequency. However, in the 2000s a power wall was hit, and the exponential frequency growth stopped at just below 4.0GHz [13]. GPUs on the other hand are still growing in performance due to massive parallelism and they typically have 10 to 100 times more cores and work fine with lower frequency (400MHz). As it has been discussed before using parallelism (scaling out) it seems that the performance could be increased infinitely. Important thing to notice again is that increased parallelism will only increase the performance of parallel code sections, meaning that the serial part of the code will become the bottleneck in the future (Amdahl's law.) [13]

It can be deduced that the best practice would be a combination of massively parallel GPU and a fast multi-core CPU.

The main vendors for GPU are Intel, AMD and Nvidia. However, Intel has focused in the integrated and low-performance market and for high-performance and discrete graphics, AMD and Nvidia are the sole two suppliers. There are three languages suitable for NVIDIA GPUs: Nvidia CUDA, DirectCompute and OpenCL [13]. OpenCL can also be used with AMD accelerated parallel processing.

2.7.1 NVIDIA CUDA

One way to accelerate the computation for, e.g., general Python program is using CUDA programming. This kind of programming is supported in NVIDIA GPUs. Pure Python code is possible to transform to be executed in GPU for improved runtimes [31]. With CUDA the procedure is as simple as telling that there is a function that we want to parallelize. The function is then automatically compiled and moved to GPU. This is done with Numba package's 'vectorize'-decorator. It can be seen that in simple functions most of the time is spent on transferring data between host and the GPU. In complex cases, it is generally desirable to offload more computationally intense functions to the GPU to reduce the amount of data transfer.

2.7.2 Modern GPU architecture

GPU is composed of hundreds of simple cores that are able to handle thousands of concurrent hardware threads. GPUs are designed to maximize floating-point throughput. GPUs based on the Fermi architecture feature up to 512 accelerator cores called CUDA cores. Each CUDA core has a fully pipelined integer Arithmetic Logic Unit (ALU) and a floating point unit (FPU) that executes one integer or

floating point instruction per clock cycle. The CUDA cores are organized in 16 streaming multiprocessors, each with 32 CUDA cores [13]. This naturally is the case just for the current generation of Nvidia graphic cards.

The massively threaded architecture of the GPU is used to hide memory latencies. Even though GPU has a superior memory bandwidth compared to CPUs, it can take hundreds of clock cycles to start the fetch of a single element from main GPU memory [13]. The latency is automatically hidden by the GPU through rapid switching between threads. Once a thread stalls on a memory fetch, the GPU instantly switches to the next available thread.

Nvidia has also released a new architecture named Kepler, which differs from the standard Fermi architecture in some ways. Kepler has dynamic parallelism which enables a GPU kernel to dynamically launch other kernels or itself recursively. Furthermore, multiple CPU cores can launch job on a single GPU simultaneously. Kepler architecture has only four multiprocessors, but each with 192 CUDA cores, which totals to 1536 CUDA cores for one chip, compared to 512 for the Fermi architecture. Also, the clock frequency has been decreased from 1.5 GHz to just over 1 GHz. Kepler architecture has a lower power consumption, yet it doubles the gigaflops performance [13].

3 Tools & Platforms

In this chapter I present and review technologies, that can be used for parallel programming. These technologies include tools, that can be used on cloud platforms, as well as tools, that can be used on local machine.

3.1 Cloud Platforms

The three most commonly used cloud platforms are AWS (Amazon Web Service) [2], Microsoft Azure [27] and Google Cloud [18]. All of them have similar features, such as storage, computation and analytics solutions, but in this thesis I mainly look into how AWS does cloud platform services. As was mentioned before, the different cloud platforms offer fairly similar section of features, and because it is more relevant for the topic of this thesis to develop the implementation on local machines, I decided to review only one cloud platform and chose one that I prefer personally and want to get better understanding of its tools.

3.2 AWS Lambda

At AWS Lambda commercial site [32], it is stated that it is an adaptation of the lambda architecture design pattern, which combines both batch and stream processing capabilities for online processing and handling massive data volumes [24]. AWS Lambda is a proficient tool for 'on-the-fly' optimized online data analysis. The lambda design pattern is suited to applications, where there are time delays in data collection as well as where data validity is required for online processing, as it arrives [24].

AWS Lambda allows implementing microservice architectures without the need of managing servers. As a result, it facilitates the creation of functions that can be easily deployed and automatically scaled and it also helps reduce infrastructure and operation costs. One usage example is to use AWS Lambda, e.g., for online sensor data to find efficient solutions to process large data sets [32].

Functions in AWS Lambda can be developed in Java 8, Python or Node.js. Unlike microservice architecture, where both gateway services are implemented in the same web application, in AWS Lambda the two gateway services have to be implemented as two separate functions that receive requests from end-users through the Internet, execute the microservices through REST and return the results to end-users [45].

AWS Lambda, which is exclusively designed to deploy microservices at a more granular level, allows companies to reduce their infrastructure costs up to 77.08% [45].

In the AWS commercial site it is also promised, that with AWS lambda there are no servers to manage, as AWS Lambda automatically runs the core without requiring provision or server management. AWS Lambda also scales continuously, meaning that the application is scaled by running code in response to each trigger. The code

runs in parallel and processes each trigger individually, scaling precisely with the size of the workload. Furthermore, the pricing of the service is done for every 100ms the code is executed and the number of times the code is triggered [32].

AWS Lambda can be used for data processing, both real-time file processing and real-time stream processing for cases that process real-time streaming data for, for example, application activity tracking, transaction order processing and metrics generation [32]. Real-time file processing is used for example, to process data immediately after an upload.

3.3 Apache Hadoop

The programming model in Hadoop [4] is MapReduce. Apache Hadoop is an open software framework for storing and processing large datasets using clusters of commodity hardware. It is designed to scale up with thousands of nodes and it is highly fault tolerant [40]. As a result of the master-worker computing model, Hadoop is an efficient and fast solution of Big Data processing, and its distributed data storage can process Big Data even to PB scale [47]. One major benefit of Hadoop is that it automatically maintains multiple file copies that can be redeployed during the data processing in case of failure. Hadoop consist of several components, the main components being HDFS and YARN, which is a resource management layer and takes care of scheduling jobs across the cluster. Other components are Hadoop MR, Hive, Pig, HBase and Storm (See Figure 7) [40].

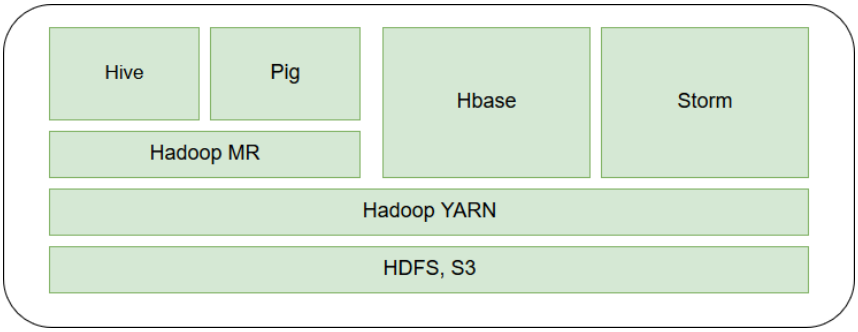

Figure 7: Different components in Hadoop Stack [40]

Hadoop MapReduce is a software framework for easily writing applications which process large amounts of data in-parallel on large clusters consisting of thousands of nodes of commodity hardware in fault-tolerant manner [5]. Hive can be used for reading, writing and managing large datasets residing in distributed storage using SQL [7]. HBase, on the other hand, is used for random lookups from big data. Analysis for large data sets can be done with Apache Pig [8]. Apache Storm is a distributed computation system that can be used, for example, for continuous computation and online machine learning [9].

3.3.1 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is, as the name suggests, a distributed file system, which is used as the primary storage for Hadoop cluster. HDFS cluster consists of two types of nodes: NameNode and DataNode [10]. The NameNode manages the file system namespace, maintains the file system tree and stores all the metadata. DataNodes act as the storage system for the HDFS files.

3.4 Apache Spark

Apache Spark is a powerful tool that combines an engine for distributing programs across clusters of machines. It is arguably first open source software that makes distributed computing accessible to data scientist [38]. The predecessor of Apache Spark is MapReduce, which is underlying mechanic in Apache Spark. The essential mechanic in MapReduce is the possibility to divide the execution of a task across numerous machines. The MapReduce engine achieves nearly linear scalability; When the dataset increases in size, it is possible to add more computers to compute the task in same amount of time. MapReduce is also resilient as failures that seldom occur on a single machine occur all the time on the clusters of thousands. It breaks up work into small tasks and can accommodate task failures without endangering the job [38].

Apache Spark is an alternative for Apache Hadoop and it is designed to overcome the disk I/O limitations and improve the performance of earlier systems. For certain tasks Spark is tested to be 100x faster than Hadoop MapReduce when data can fit into memory and 10x faster when data resides on disk. Spark can read data from HDFS and run on Hadoop YARN manager [40].

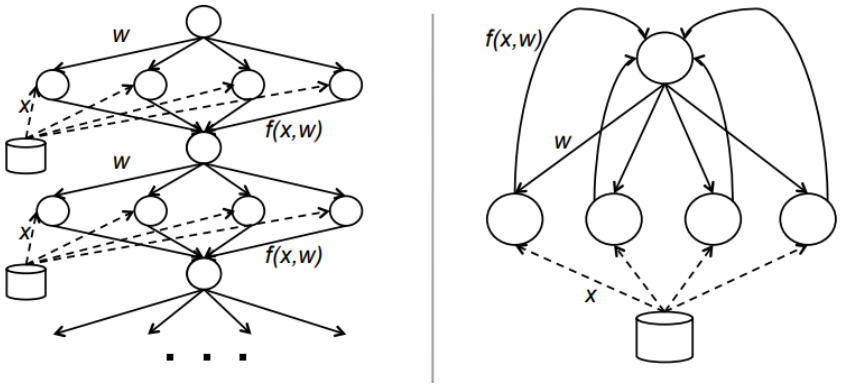

Figure 8: Data Flow of logistic regression with Hadoop (left) vs. Spark (right). Solid lines show data flow within the framework. Dashed lines shows reads from a distributed file system. Spark reuses in-memory data across iterations to improve efficiency [22]

Hindman et.al. noticed that with Hadoop each iteration takes 127 seconds on average, because it runs as a separate MapReduce job. On the other hand with Spark the first iteration takes little longer, approximately 174 seconds, but subsequent iterations only take about 6 seconds, leading to a speedup of up to 10x for 30 iterations (See Figure 9). This happens because the cost of reading the data from disk and parsing it is much higher than the cost of evaluating the gradient function computed by the job on each iteration [22]. Hadoop incurs the reading/parsing cost on each iteration, while Spark reuses cached blocks of parsed data and only incurs this cost once.

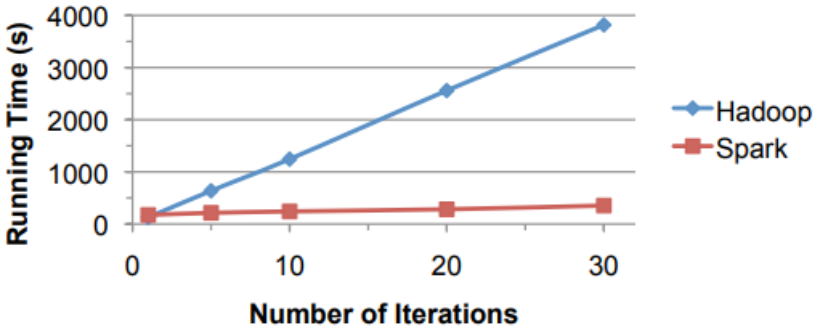

Figure 9: Hadoop and Spark logistic regression running times [22]

Linear scalability and fault tolerance are naturally embedded in Spark, but it also extends them in three important ways. Instead of rigid map-then-reduce format, Spark solves the cyclic problem that MapReduce has, with engine that can execute Directed Acyclic Graph (DAG) of operators. Unlike MapReduce, which writes out intermediate results to the distributed filesystem, Spark passes them directly along the pipeline [38]. The second extension of Spark is that it complements the capability to express complex pipelines in a few line of code. Lastly, Spark is equipped with in-memory processing. The Resilient Distributed Dataset (RDD) abstraction enables developers to materialize any point in a processing pipeline into memory across the cluster [38]. This brings out the major benefit of Spark; if same data is needed in future steps, it is not required to recompute it or reload it from disk. This feature enables opportunities that distributed processing engines could not previously achieve.

The forementioned RDD is an abstraction of read-only collection of records partitioned across the distributed computer nodes in memory [47]. The dataset is resilient because the data persists in memory or disk and if a portion of the dataset is lost, it can be rebuilt [10]. The reasoning behind RDDs is to process Big Data with various formats interactively with low latency on the premise of keeping the whole system fault-tolerant. RDD objects can be manipulated during the Spark computation and the intermediate results are stored generally in volatile memory also as RDD objects that can be reused [47]. The two types of operations for RDD objects are transformations and actions. Transformations create a new altered RDD object

from the current RDD object and return it. Because RDD objects are immutable in nature, the only way to modify them is to create a new RDD. Some examples for transformations are: map, filter and reduceByKey. Actions on the other hand work with the existing RDD object and/or extract information from it, e.g, count, foreach and takeOrdered. When action is triggered after the result, new RDD is not formed. The values of action are stored to drivers or to the external storage system. Unlike transformations, which return RDD objects, actions return a value from the RDD to the main program.

One very important, perhaps the most important, merit of Spark is not any functional property, but the grouping of the pipeline into single programming environment. The speed increase in development should not be underestimated.

Here is an example of a simple Spark application written in Scala that reads given text file, counts the words in it and prints the wordcounts.

Listing 1: Example Scala Spark MapReduce Code

```
val conf = new SparkConf().setAppName("Spark WordCount 1")
val sc = new SparkContext(conf)
val textFile = sc.textFile("path/to/textfile.txt")
val wordCounts = textFile.flatMap(line => line.split(" "))
                         .map(word => (word, 1))
                         .reduceByKey(_ + _)
System.out.println(wordCounts.collect().mkString(", "))
```

Here is another implementation that does the same but utilizes more shorthand operations.

Listing 2: Example Scala Spark MapReduce Code with Shorthand Operations

```
val conf = new SparkConf().setAppName("Spark WordCount 2")
val sc = new SparkContext(conf)
val tokenizedTextFile = sc.textFile("path/to/textfile.txt")
                          .flatMap(_.split(" "))
val wordCounts = tokenizedTextFile.map((_, 1))
                                  .reduceByKey(_ + _)
System.out.println(wordCounts.collect().mkString(", "))
```

Spark also utilizes caching, which can be tricky to use, as caching data doesn't automatically guarantee improvement in execution time. Nevertheless, in-memory caching makes Spark ideal for iteration both at the micro and macro level in machine learning algorithms. At micro level iterations, the hyperparameters are being tuned, and at macro level iterations, the problem is being solved [44]. Caching should be used for example in machine learning algorithms that make multiple passes over the dataset [38].

In short Spark is just a processing framework, which works on distributed collections, these collections are partitioned and the number of partitions are defined from

the source. The collections are lazily evaluated meaning that nothing is done until the final result is requested. In practice, one only writes the instructions for what spark has to do. The collections of what one is working with are RDDs and DataFrames [21]. RDDs are actually just collections of java objects, which are JVM objects. These objects are slowest, but most flexible. DataFrames are mainly tabular data that are able to do structural data, but it is not straightforward. DataFrames have fast serialization/deserialization and have compact and fast memory management. In addition to that, DataFrames are also compatible with SparkSQL. SparkSQL is used for data queries from database or filesystem. In addition, DataFrames have generally faster transformations than DataSets and RDDs. Spark offers a distributed file system with failure and data replication management and it allows the addition of new nodes at runtime. In addition, Spark provide a set of tools for data analysis and management that is easy to manage [36].

How to use Spark?

The most common programming languages that support Spark are Python, Scala and Java. In this thesis, I concentrate implementing solutions with Scala and Python.

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the main program called the driver program. On a cluster, SparkContext can connect to several types of cluster managers [10]. The cluster manager allocates resources across applications and the executors on the worker nodes are acquired to run computations and store data for the application. Lastly, SparkContext sends tasks for the executors to run. At the end of the execution, actions are executed at the workers and results are returned to the driver program [10].

There are different approaches of how to deploy Spark applications. The three main ways to run Spark are standalone, with Hadoop Yarn or with Mesos.

3.4.1 Standalone

Spark Standalone cluster is Spark's own built-in cluster environment. It is the easiest way to run Spark applications in a clustered environment in many cases, because Spark Standalone is available in the default distribution of Apache Spark. Spark Standalone contains Standalone Master and Standalone Worker/Slave. Standalone Master is the resource manager for the Spark Standalone cluster and the Standalone Worker is consequently the worker in the cluster. In Standalone cluster mode, Spark allocates resources based on cores. By default all the cores in the cluster are used. However, Standalone only allows one executor be allocated on each worker per application.

3.4.2 Hadoop Yarn

Spark on YARN supports multiple application attempts and supports data locality for data in HDFS. In addition, it is possible to take advantage of Hadoop's security

and run Spark in a secure Hadoop environment [25]. Spark application deployed to YARN is a YARN-compatible execution framework that can be deployed to a YARN cluster. When deploying the application in this manner, the Spark executor maps to a single YARN container.

3.4.3 Mesos

Mesos is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks [22]. The reasoning behind sharing is that it improves cluster utilization and avoids per-framework data replication. Key feature of Mesos is a distributed two-level scheduling mechanism also known as resource offers. Basically Mesos decides the amount of resources to offer each framework and in turn frameworks decide, which resources to accept and which computations to run on them. In a study composed by Hindman et al., the results show that Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks. In addition to that, Mesos can scale to 50,000 emulated nodes and is resilient to failures [22].

The advantages of deploying Spark with Mesos are dynamic partitioning between Spark and other frameworks and scalable partitioning between multiple instances of Spark. With Mesos it is possible to create specialized frameworks optimized for workloads for which more general execution layers may not be optimal. Jobs that perform poorly on Hadoop, e.g., iterative jobs, where dataset is reused across number of iterations, can be improved by using simple specialized framework like Spark. In their study, Hindman et al. compared two implementations of logistic regression, an algorithm used in machine learning. One implementation was with Hadoop and the other with Spark. In Hadoop, the data must still be reloaded from disk at each iteration. Spark, however, uses the long-lived nature of Mesos executors to cache a slice of the dataset in memory at each executor and the run multiple iterations on this cached data. With Spark built on top of Mesos Hindman et al. outperformed Hadoop by 10x for iterative jobs [22]. Another benefit of using Mesos's API was that it saved the time to write a master daemon, slave daemon and communication protocols between them for Spark.

3.5 Database Integration

While selecting the deployment environment for Spark application is a crucial task, it is not the only important choice one has to make while developing Spark applications. Because the applications are more often than not using big data, a suitable data storage must be used to handle large amounts of data. In his blogpost, Ion Stoica reminds that Spark is not intended to replace, but to enhance the forementioned Hadoop Stack [41]. Spark is essentially designed to not only work with HDFS, but also with other storage systems, such as HBase and Amazon's S3.

3.5.1 HBase

HBase is a data store that runs on top of HDFS. In HBase, data is stored in tables, which have rows and columns. This terminology is misleading as it overlaps with relation databases (RDBMSs), while HBase falls under NoSQL category, which means that it does not support structured query languages (SQL). In HBase reference book, it is said that HBase can be thought as a multi-dimensional map instead [6]. For storing data, HBase uses Hfiles, that are similar to HDFS files, for low latency data fetching.

Hbase is very good solution when we have hundreds of millions or billions of rows to work with. If the amount is under hundred million rows, then the solution should be a traditional RDBMS such as MySQL/PostgreSQL. This is due to the possibility of all data residing on a single node while the rest of the cluster is sitting idle [6]. Furthermore, when using HBase it is compulsory to make sure that it is possible to develop the applications without the features of RDBMS. Finally, enough hardware is needed to get all the benefits from the distributed file system.

HBase is usually compared against plain HDFS. The main difference is in their ability to store large files and fast individual record lookups on those files. HDFS is more suitable to store large files, but the lookups are much slower than with HBase. In other words it depends whether a full scan of the data or collection of particular records are needed in the application. When writing data from Spark to HBase it is not worthwhile to write it to database directly, instead data is written to HFiles which can be bulk imported.

3.6 Hosting Spark Applications

When hosting a Spark Application, one has not only to take account the scalability and performance but also the pricing of the service. One very competent hosting service is Amazon Web Service. Amazon offers large variety of different containers to host services, so some research is needed to be able to decide the best suited option for different applications. As we can see in Figure 10. the amount of services Amazon offers can be overwhelming

For instance one recent release from Amazon is AWS Fargate, which provides ease-of-use, as containers can be run without managing servers or clusters. With AWS Fargate it is no longer required to provision, configure, and scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale clusters, or optimize cluster packing [11]. This means that with AWS Fargate the only thing that needs to be considered is the containers, so one can focus on building and operating the application. Also, scaling is more fluent as it is not needed to ponder about provisioning enough compute resources for container applications. After the requirements are defined AWS Fargate manages all the scaling

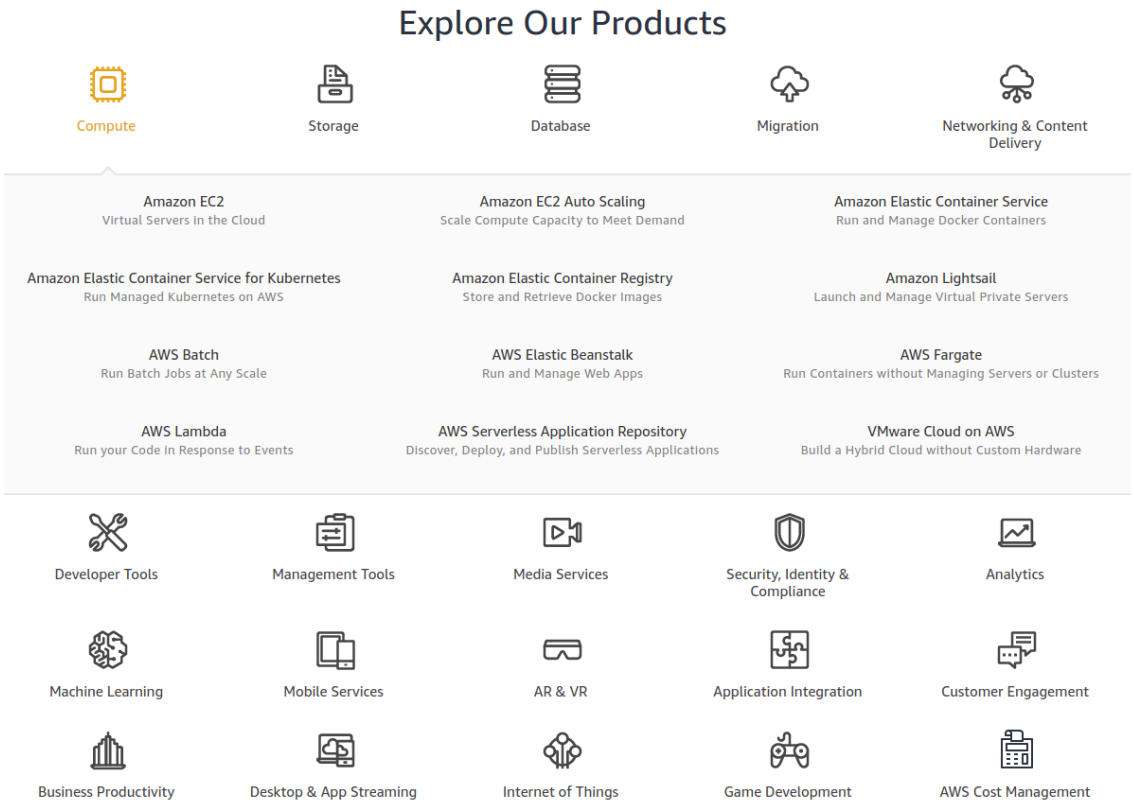

Figure 10: List of AWS Products [2].

and infrastructure needed to run containers in a highly-available manner.

3.6.1 Amazon Elastic Map Reduce

When deciding platforms to run Spark applications on, one viable option to take into account is Amazon Elastic Map Reduce (AWS EMR). Amazon EMR provides a managed Hadoop framework that enables easy, fast, and cost-effective way to process vast amounts of data across dynamically scalable Amazon EC2 instances. It securely and reliably handles a broad set of big data use cases, including log analysis, data transformations (ETL), and machine learning [20]. Hadoop clusters running on Amazon EMR use EC2 instances as virtual Linux servers for the master and slave nodes. Amazon S3 can be used for bulk storage of input and output data, furthermore with the data stored in S3 it is possible to have multiple clusters accessing the same data simultaneously. In addition it is possible to integrate CloudWatch to monitor cluster performance and raise alarms. On April 10, 2018 Amazon EMR 5.13.0 was released, which provides support for Spark 2.3.0 which means it is possible to take advantages on Spark's most recent features such as Structured Streaming.

When using Amazon EMR it is beneficial to use it with Amazon EC2 Spot Instances, which offer spare compute capacity available in the AWS cloud at steep

discounts compared to On-Demand instances. Spot Instances enables optimization of costs on the AWS cloud and scaling for the application's throughput up to 10 times for the same budget [1]. The upside of Spot Instances are that they can be interrupted by EC2 with two minutes of notification when EC2 needs the capacity back. They can be used for various fault-tolerant and flexible applications, for instance applications that work with big data, containerized workloads and high-performance computing (HPC). Spot instances can utilize other AWS services such as EMR, Auto Scaling, Elastic Container Service and CloudFormation. Spot Instances can be used with Spot Fleet, which is used to automate the management of the instances [1].
At the official AWS website the listed benefits of Spot Instances are among other things reduced costs, scalable applications and Hibernate & Stop-Start features.

The steps needed to take to start using Spark on Amazon EMR are not too challenging. First step is to create an EMR cluster with Spark. This can be done in the Amazon EMR console or by using the AWS CLI. In this step, it is required to determine the Amazon Release Version, EC2 instance types and count, and the default roles of users. You can test your Spark code immediately by accessing the Spark Shell in EMR. First connect to the EMR master node via SSH and start the Spark Shell. To run Spark applications on EMR cluster you can configure steps that will run the application. In the step it is required to determine the path to the source JAR, which can be stored to S3 storage. In addition the deploy mode must be chosen and any additional spark-submit options for example '–class' [20]. It is important to notice that the basic settings deploy 3 instances of m4.large EC2 containers that are On-Demand, so to avoid unnecessary bills Spot Instances should be integrated to hibernate the containers when calculations are not happening.

Additional Spark tools

While the main parts of Scala are inside Scala Core, which refers to to the distributed execution engine and the core of Spark APIs, there are also additional tools inside Spark. While features in Spark Core are stable and compatible, the additional features are in alpha or beta phase, meaning that they are likely to undergo some changes as they mature [38].

The main subprojects in Spark are SparkSQL, Spark Streaming, MLlib, GraphX. I will next go through them briefly

MLlib provides a set of machine learning algorithms on top of Spark. MLlib represent data as Vector objects and contains some linear algebra functionality for operating on Matrix objects.

Spark also has an in-built SQL-query engine called SparkSQL. It works both with datasets stored in HDFS and with existing RDDs.

Furthermore Spark contains subproject named GraphX, which is a tool for graph

processing. Graph is a term in computer science for structure that consists of set of vertices that are connected to each other with set of edges. In GraphX however graphs are represented with a pair of RDDs; RDDs of vertices and RDDs of edges. This makes GraphX highly declarative, and a basic PageRank algorithm can be represented with just a few lines of code [38].

3.6.2 Spark Streaming & Structured Streaming

Spark Streaming is a tool for processing the data continuously. While Spark's typical process operates on the dataset at once, Spark streaming aims for low-latency; as data becomes available it needs to be transformed and dealt with in near real time [38]. As this thesis aims for continuous calculations of dynamic prices, it is most likely that Spark Streaming will aid in this task. However, the time this thesis is being written, Spark 2.3.x has been published, and with it came Structured Streaming, which improves Spark Streaming multiple ways.

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. Structured Streaming is used to express streaming computation the same way that a batch computation on static data is expressed. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

Internally, Structured Streaming queries are processed using micro-batch processing engine, which processes data streams as a series of small batch jobs achieving end-to-end latencies as low as 100 milliseconds and exactly-one fault-tolerance guarantees. Since Spark 2.3 a new low-latency processing model called Continuous Processing has been introduced, that can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.

3.6.3 Sparkling Water

While developing applications on Spark that require complex machine learning algorithms sometimes the integrated Mllib doesn't have all the necessary features. For example some regression and classification models (e.g., Random Forest) only accept numeric values. One considerable alternative is H2O, which is in-memory, distributed, fast and scalable machine learning platform. Sparkling Water in turn combines these scalable machine learning algorithms of H2O with the capabilities of Spark.

To access H2O, the driver program in Sparkling Water creates a SparkContext which is used to create and H2OContext, which in turn is used to start H2O services on the Spark executors. The created H2O Context is a connection to H2O cluster and enables communication between H2O and Spark. To properly use H2O's algorithms, the RDDs needs to be converted to H2O's distributed data frame called H2OFrame. This is due to the difference in data layout between Spark (blocks/rows) and H2O

(columns). However, the overhead of making data copy is low as the data is stored in H2O in a highly compressed format.

With Sparkling Water one can not only write own applications, but also utilize the H2O Flow UI. Usually, Sparkling Water is used to improve existing Spark applications where the need of advanced machine learning algorithms rises. Typical use case would be using Spark to fetch and transform the data and the resulting table is passed to the H2O Machine Learning algorithm. Sparkling Water can be directly put to use with Spark application written in Scala. In addition, integrations for R and Python are also available, called RSparkling and PySparkling.

Scala or Python?

Major benefits of using Scala comes from the fact that Spark is written in Scala. Firstly, this reduces performance overhead. When running an algorithm in other languages, for example Python, on top over JVM-based language like Scala, some work has to be done to pass code and data across different environments, which can lead to losses in translation. Second benefit of using Scala is that it gives access to all Spark's machine learning, stream processing and graph analytics libraries, which again are written in Scala [38]. All of these features will eventually be available to other languages, but to make sure all features are supported, Scala is a safe choice. Lastly, using Scala will help to understand the design choices of Spark altogether.

Programming Spark with Python can be done with the Spark Python API (PySpark). Advantages of using Python are in the simplicity of its implementation and the support of Python libraries. Another benefit is the support for CUDA programming. If the problem requires both parallel data processing and intensive GPU calculations, it is possible to implement both Spark and CUDA codes with Python. In addition, unlike Scala, it is not needed to compile Python code before you can run it.

Although I have done majority of my programming with Python, it seems like an obvious choice. However, because, as previously mentioned, benefits of Scala are numerous, thus I am going to use both Python and Scala for the implementations.

3.7 PyTorch

PyTorch is a recent (in early-release beta phase) Python package that provides tensor computation with strong GPU acceleration and Deep Neural Networks built on a tape-based autograd system, meaning that it uses reverse-mode automatic differentation [34], which will be explained later. Neural Network is used to process complex data inputs into a space that computers can understand [17]. Deep Neural Network is a neural network that has more than two layers. PyTorch is compatible with NumPy, SciPy and Cython so they can be used to extend PyTorch when needed. Usually, PyTorch is used as a replacement for NumPy to use the power of GPUs or

as a deep learning research platform that provides maximum flexibility and speed [34].

PyTorch provides Tensors that can live either on the CPU or the GPU, and accelerate computations by a huge amount. One major feature in PyTorch is its unique way of building neural networks: it uses and replays a tape recorder. Unlike most frameworks, such as TensorFlow, that have a static view of the world, PyTorch uses a technique called Reverse-mode auto-differentation [34], which allows changes to the way the network behaves arbitrarily with zero lag or overhead. Thus, changing the way the network behaves, doesn't mean that one has to start from scratch. While other implementation use the technique, PyTorch is one of the fastest implementations and it proved the best of speed and flexibility.

3.8 TensorFlowOnSpark

Traditionally, separate clusters are needed, when developing application handling both big data and machine learning algorithms. This was also the case with the deep learning network TensorFlow; dedicated clusters were needed, even though TensorFlow introduces HDFS support in 2016, as TensorFlow programs could not be deployed on existing big data clusters [49]. This lead to the implementations of Tensorflow with Spark clusters. However, TensorFlow processes were unable to communicate with each other directly, which would result to significant effort in migrating existing TensorFlow programs.

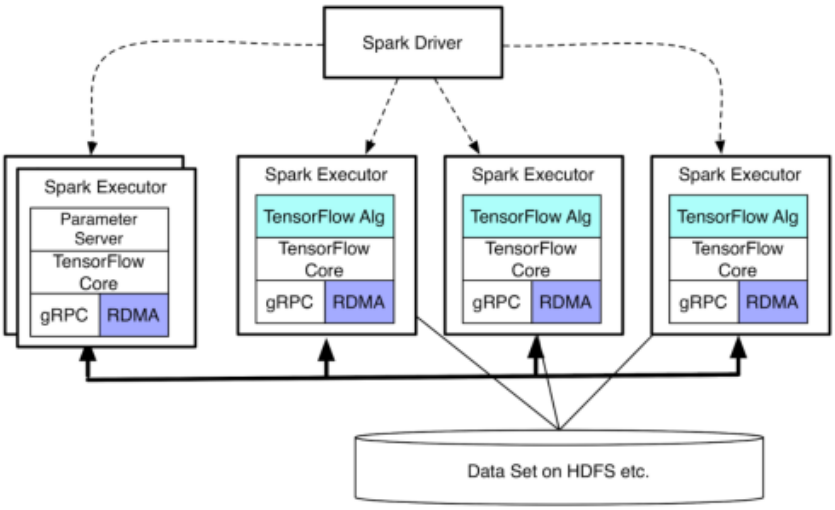

Figure 11: TensorFlowOnSpark system architecture [49]

TensorFlowOnSpark framework was introduced for a response to these problems. It enables distributed TensorFlow execution on Spark and Hadoop clusters. Because process-to-process direct communication is supported in TensorFlowOnSpark, programs taking advantage of it can be scaled easily by adding more machines [49]. Other benefits are easy migration of existing TensorFlow programs and support for

TensorFlow functionalities. In addition, datasets on HDFS or other sources can be pushed by Spark or pulled by TensorFlow [43].

Two different modes are provided in TensorFlowOnSpark to ingest data for training and interference: TensorFlow QueueRunners and Spark Feeding. QueueRunners as well as TensorFlow's file readers are used to read data directly from HDFS files without the involvement of Spark. On the other hand, Spark RDD data is fed to each Spark executor, which feeds the data into the TensorFlow graph with feed_dict [49].

3.9 Selecting tools for the problem

When selecting the tool to handle problems when working with Big Data, there are several aspects to take into account. One approach is to figure out what the algorithm/application requirements are. Common questions to ask are: How quickly do we need the result, how big is the data to be processed and does the model building require several iterations or just one iteration?

As the previous issues are application/algorithm dependent, they need to be solved before approaching system level problems [40]. One has to ponder, if there will be need for more data processing ability in the future, is the data transfer rate critical for this application and is it needed to handle hardware failures within the application.

A detailed comparison of different platforms has been implemented by Singh et al. which rates particular characteristics of platforms. The first three characteristics, scalability, data I/O performance and fault tolerance are system/platform dependent. The last three characteristics, real-time processing, data size support and the support for iterative tasks, are on the other hand application/algorithm dependent [40].

Scaling type	Platforms (Communication Scheme)	System/Platform			Application/Algorithm		
		Scalability	Data I/O performance	Fault tolerance	Real-time processing	Data size supported	Iterative task support
Horizontal scaling	Peer-to-Peer (TCP/IP)	★★★★★	★	★	★	★★★★★	★★
	Virtual clusters (MapRedce/MPI)	★★★★★	★★	★★★★★	★★	★★★★	★★
	Virtual clusters (Spark)	★★★★★	★★★	★★★★★	★★	★★★★	★★★
Vertical scaling	HPC clusters (MPI/Mapreduce)	★★★	★★★★	★★★★	★★★	★★★★	★★★★
	Multicore (Multithreading)	★★	★★★★	★★★★	★★★	★★	★★★★
	GPU (CUDA)	★★	★★★★★	★★★★	★★★★★	★★	★★★★
	FPGA (HDL)	★	★★★★★	★★★★	★★★★★	★★	★★★★

Figure 12: Comparison of different platforms (along with their communication mechanisms) based on various characteristics. [40]

In Figure 12 this comparison is visualized. It should be noted that the comparisons are qualitative and not quantitative between the platforms. For example a two stars rating should be interpreted as better than having a one star rating and does not necessarily mean that it is two times better than the platform with one star rating [40].

4 Gaming Industry

4.1 Big Data in Gaming industry

Big data has had a huge impact in gaming industry. The ability to understand how to benefit from collected data for monetization and optimization can lead to significant advantages. Reality Games have gathered that their userbase, 2 billion gamers, produce approximately 50Tb of data/day. Their AAA multiplayer titles produce 1 Tb of data/day from in-game telemetry and from social games about 150 Gb of data is produced each day [35]. This data can be used to gain understanding what the users want by monitoring the behavior of users.

4.2 Monetization models

In his book Eri Ries writes about the Three Engines of Growth, which are the Sticky Engine of Growth, The Viral Engine of Growth and The Paid Engine of Growth [37]. These models are designed to give businesses a set of metrics on which to focus their energies. In short, the Sticky Engine of Growth focuses on retaining customers for the long term, The Viral Engine of Growth is the domain of word of mouth and having the product advertise itself and The Paid Engine of Growth mean that each customer needs to yield profit.

In gaming industry the examples for these models (Sticky, Viral, Paid) are subscription based games, such as World of Warcraft, games that have gone 'viral' (Flappy Bird), and Premium Games/AAA titles, for example Call of Duty. In this thesis, the interest lies in the monetization models of two categories in the Paid model; Premium and Freemium games. Premium games references to games that require initial payment before you gain access to the game. Basically games that you would buy from a vendor. Freemium, or free-to-play is a business model that lets people use games, software etc. free of charge, but there usually is a payment charged for additional features. With Freemium model the maximum number of users is achieved, from which small portion (10-20%) are willing to buy additional products. In his book, Eric Seufert argues that the Freemium model has become the dominant means of generating revenue on mobile devices [39].

For Freemium games there are basically two options: In-Game-Ads or In-App-Purchases (IAPs). With big data, companies could even create meaningful marketing messages for each player, meaning personalized ads. Ads are easy way to get some revenue, but having an ad pop up in the screen while playing a game can have negative impact to the gaming experience. The most efficient ad is so-called Rewarded Ad, which gives the player some kind of reward, such as in-game soft-currency.

The other option is In-App-Purchases. These are the major source of revenue in Freemium games. IAPs can be something that enhance the gaming experience or give advantage on other players. IAPs are a great monetization tool granted that there is

data behind them [35]. With Freemium games developers have the opportunity to track behavior of a new customer right away and see where they are hitting roadblock and stop playing or what drives them to make purchases. Using big data in Freemium stage allows game developers to accurately measure, predict, and track player behavior to optimize the experience, increasing the likelihood users converting to a paid model.

To compare the revenue from Premium- and Freemium-models, with Premium-model the game developer receives constant sum for each player, but with Freemium games, although the players who don't pay anything will be shown ads, the paying users can bring thousands of dollars of revenue.

Despite it is important to have an appealing game, in this thesis we do not dwelve in game design nor plan to make adjustments to the games themselves, but rather only to the pricing of their IAPs. This is done with a Dynamic Pricing Engine that calculates the price elasticity of demand. These topics will be discussed in the next sections.

4.3 Price Elasticity of Demand

Price elasticity of demand derives from the basic law of demand in economics. It is general knowledge that the higher the price of a products is, the less consumers will purchase it. Price elasticity of demand is simply the proportionate change in demand given a change in price. For example, if a one-percent drop in the price of the product produces a one-percent increase in demand for the products, the price elasticity of demand is said to be one [3].

4.3.1 Price Elasticity of Demand with Formulas

Generally price elasticity can be presented as

$$\frac{\Delta Quantity}{Quantity} / \frac{\Delta Price}{Price} = \%\Delta Quantity / \%\Delta Price \tag{2}$$

where delta is the absolute change. Two common ways to discover price elasticity of demand are arc elasticity and point elasticity. Arc elasticity is mostly used with economists. Point elasticity removes some of the problems faced when using arc elasticity. The formula for arc elasticity is

$$E_p = \frac{Q_2 - Q_1}{(Q_1 + Q_2)/2} / \frac{P_2 - P_1}{(P_1 + P_2)/2} \tag{3}$$

where
E_p = Coefficient of price elasticity
Q_1 = Original quantity demanded
Q_2 = New quantity demanded
P_1 = Original price

P_2 = New price

The formula for point elasticity is

$$\varepsilon_p = \frac{dQ}{dP} * \frac{P_1}{Q_1} \tag{4}$$

where
ε_p = Coefficient of point elasticity
dQ/dP = Derivative of Q with respect to P

When using linear function the derivative equals the slope of the demand curve, and the formula becomes

$$\varepsilon_p = \frac{\Delta Q}{\Delta P} * \frac{P_1}{Q_1} \tag{5}$$

ΔQ = Absolute change in quantity
ΔP = Absolute change in price

When the demand curve is linear it is possible to use the function of point elasticity to calculate values for price and quantity where $\varepsilon_p = -1$

$$\varepsilon_p = (dQ/dP) * (\frac{P_{optimal}}{Q_{optimal}}) = -1 \tag{6}$$

$$P_{optimal} = -Q_{optimal}/(\frac{dQ}{dP}) \tag{7}$$

And in case of linear point elasticity where the first derivative equals the slope as mentioned earlier, we get $P_{optimal} = -Q_{optimal}/\frac{\Delta Q}{\Delta P}$

4.3.2 Regression Analysis

Regression analysis is possible to do with single linear regression, single polynomial regression, second degree polynomial regression, multiple regression and exponential regression.

Single Linear Regression

Regression analysis is used to determine the demand curve. Single linear regression is done by fitting a linear line to the price/quantity points. With a single linear regression the linear equation of the demand line is of the following format: $Y = aX + b$. This means that the format for the optimal price is

$$P_{optimal} = aQ_{optimal} + b \tag{8}$$

The equation of the point elasticity ε_p is

$$\varepsilon_p = (dQ/dP) * (P_1/Q_1) \tag{9}$$

where
(dQ/dP) = derivative of the demand function.

With linear regression the demand function is known and the derivative is known. The point elasticity can then be calculated as follows
$P = aQ + b$
$Q = (P - b)/a = P/a - b/a$
$dQ/dP = 1/a$
$\varepsilon_p = (dQ/dP) * (P/Q)$
$\varepsilon_p = (1/a) * (P/Q)$
$\varepsilon_p = (1/a) * \frac{P}{(P-b)/a}$ <=>
$\varepsilon_p = P/(P - b)$

The optimal price can then be calculated by finding the value $Q_{optimal}$ where point elasticity $\varepsilon_{optimal} = -1$. Using the point elasticity calculated form the known regression curve the optimal price can be calculated
$\varepsilon_{optimal} = P_{optimal}/(P_{optimal} - b) = -1$ <=>
$P_{optimal} = \frac{1}{2}b$

Single Polynomial Regression

Calculating the point elasticity from the second degree polynomial doesn't differ from the single regression calculations. The point to note is on the dQ/dP, i.e., the derivative of Q with respect to P, i.e., the derivative of equation used to get quantities of prices, which differs when dealing with polynomial regressions. This makes calculating the elasticity more challenging.

Second Degree Polynomial

For simplicitys sake the elasticity is calcualted on the regression using point (P_j, Q_j) instead of (Q_j, P_j) ie. the Q is on the Y axel instead of X.

$\varepsilon_p = (dQ/dP) * (P_1/Q_1)$
$Q = b + a_1P + a_2P^2$
$dQ/dP = a_1 + 2 * a_2 * P$

where

a_1 = the first coefficient gotten from the regression curve
a_2 = the second coefficient gotten from the regression curve
b = the intercept of the regression curve

$\varepsilon_p = (dQ/dP) * (P_1/Q_1)$
$\varepsilon_p = (a_1 + 2 * a_2 * P) * (P_1/Q_1)$

$\varepsilon_p = (a_1 + 2 * a_2 * P) * (P_1/(b + a_1P + a_2P^2)$

The optimal price can then be calculated by finding the value $Q_{optimal}$ where point elasticity $\varepsilon_{optimal} = -1$. Using the point elasticity calculated from the known regression curve the optimal price can be calculated

$\varepsilon_p = (a_1 + 2 * a_2 * P) * (P_1/(b + a_1P + a_2P^2) = -1$
$P_{optimal} = \frac{-a_1+\sqrt{a_1^2-3*a_2*b}}{3*a_2}$

or

$P_{optimal} = -(\frac{-a_1+\sqrt{a_1^2-3*a_2*b}}{3*a_2})$

Multiple Regression

Multiple regression can be used when we want to increase the accuracy of our results by adding a second variable to the regression. This second variable should be chosen based on the product being analyzed. The variable should be something that affects the quantity sold together with the price. This can, for example, be the time of the day (night or day) or the player's level in the game. One can also choose the variable to be boolean value that determines if it's holiday season or weekend.

When working with data that has more than 64000 rows, a tool more efficient than, for example, Excel is required with multiple regression. Most data analysis tools support linear multiple regression. These tools should also provide the Coefficient of determination R^2 of the said regression providing information how well the data fit. Linear multiple regression curves is in case of pricing data often formatted as
$Q = b + a_1X_1 + a_2X_2 + ... + a_nX_n$

where
Q is the quantity sold,
b is the intercept of the regression curve,
$a_{1...n}$ are the coefficients of the variables $X_{1...n}$. These describe how much impact on the quantity the variable X_n has,
$X_{1...n}$ are the variables chosen to affect the quantity. In case of pricing data with single regression where $n = 1$, the X_1 is the price.

The calculation of the point elasticity for each variable is very similar to that of the single regression.

$\varepsilon_X = (dQ/dX) * (X/Q)$
where
dQ/dX = Derivative of Q with respect to variable X_n, i.e., the derivative of equation used to get quantities of variables. In case of linear regression, dQ/dX is the same

as a_n

For example with one extra variable the regression would be
$Q = b + a_1P + a_2X$

To get the point elasticity for price we use the known derivative $dQ/dP = a_1$
$\varepsilon_p = (dQ/dP) * (P/Q)$
$\varepsilon_p = a_1 * (P/Q)$
$\varepsilon_p = a_1 * (P/(b + a_1P + a_2X))$

Optimal price can then be calculated when

$\varepsilon_p = a_1 * (P_i/(b + a_1P_i + a_2X_i)) = -1$
$P_i = -(a_2 * X_i + b)/(2a_1)$
where
i = optimal

The optimal price is based on the effect the unknown variable X has. Because a_1 is negative (the amount of the effect price has on quantity) the optimal price is based on the variable X and its coefficient. However usually we have X already set by default, for example, the date when we want the optimal price or the creator of the product.

Exponential Regression

With exponential regression the model to use is

$$Q = a * b^P \tag{10}$$

Exponential linear regression is done by first transforming the non-linear line to linear "coordinate system" and then fitting a linear line as in normal linear regression. With a single linear regression the linear equation of the demand line is $Y = a * b^X$

The equation of the point elasticity ε_p is $\varepsilon_p = (dQ/dP) * (P/Q)$

With linear regression the demand function is known and the derivative is known. The point elasticity can then be calculated:

$P = a * b^Q$
$Q = log_b(P/a)$
$dQ/dP = 1/(P * ln(b)/a)$
$\varepsilon_p = (dQ/dP) * (P/Q) =>$
$\varepsilon_p = (1/(P * ln(b)/a)) * (P/Q) <=>$
$(1/1)/(P * ln(b)/a) * P/Q <=>$
$a/(P * ln(b)/a) * P/Q <=>$
$a * P/(Q * P * ln(b)) <==>$

$\varepsilon_p = a/(Q * ln(b))$

The optimal price can then be calculated by finding the value $Q_{optimal}$ where point elasticity $\varepsilon_{optimal} = -1$. Using the point elasticity calculated from the known regression curve the optimal price can be calculated as

$\varepsilon_{optimal} = a/(Q * ln(b)) = -1 <=>$
$Q_{optimal} = -a/ln(b)$

4.4 Apprien

In this thesis, the main target is to scale successfully an automated IAP (In-App-Purchases) pricing engine called Apprien dynamic pricing engine, which automatically calculates optimal prices utilizing price elasticity of demand. In its current state it can handle analysis of up to 100k-1M transactions, but when bigger game studios will be attracted to the product, the amount of transactions can be in the hundreds of billions. To handle the amount of traffic, some structural choices must be made. In this thesis these choices will be reviewed as follows; what technologies can be used, how do they differ from each other, what are the benefits and drawbacks of each solution. Furthermore, a prototype solution for new design will be presented, that handles the fetching and processing of transaction data.

Apprien promises to increase Life Time Value (LTV) of players with three major benefits. First benefit is higher ARPPU (Average Revenue Per Paying User) that boosts UA (User Acquisition). Secondly, with Apprien it is possible to get more precise country- and marketplace-based pricing. Finally, Apprien automatically reacts to market changes and competition.

Apprien pricing engine is a service that increases the revenue of mobile game companies and eCommerce-marketers at least by 5%. The largest verified increase in profit has been 1400%. The service calculates price elasticity of demand for different customer segments in 15 minute intervals. Thus, a company can have as many as 70 billion different prices in one month. One crucial challenge to overcome will be the amount of prices to calculate; if one customer requires calculation 70 billion prices, what will happen if we have ten thousand customers? To enable this kind of service some unconventional solutions must be utilized.

4.4.1 Dataflow

Apprien is composed of three components: ApprienGameAPI, Apprien Core (Back-end) and UnitySDK. The UnitySDK is a component that game companies have to integrate in to their game. The SDK is used to show calculated prices in the game and it communicates with the backend with ApprienGameAPI calls. When user makes a purchase, the transaction data can be directly sent by the customer or the

data is fetched from the store (Google Play for Android or App Store for iPhone) and sent to Apprien Core where the optimal price is calculated by solving the price elasticity of demand. The prices are then fetched with the UnitySDK using API requests and finally the new prices are shown in the game's internal store.

Apprien only requires the transaction data from the purchases the players make in the game, that includes quantity and price accompanied with a timestamp. From these datapoints the backend of Apprien is able to calculate new optimal price points for the app to use and as mentioned before, this can be done in intervals of 15 minutes for different segments.

The integration of Apprien to games that are developed with Unity is extremely effortless due to easy-to-use Unity SDK. However, since Apprien is used via ApprienGameAPI, the deployment of Apprien to games that are developed with any programming language that supports API requests is also straightforward.

5 Research material and methods

The research question this thesis answers is how to scale an existing application to big data scale, in other words to hundreds of millions or even billions of transaction datapoints. Experimental research is conducted to study the problem by implementing algorithms with different technologies.

In order to scale the application, it is required to do research, where the potential bottlenecks are in the current architecture. The current proven bottleneck is caused by the data transfers from database to the program. This will be investigated by comparing implementations with different methods and tools reviewed in previous sections with different amounts of data, such as 100K, 1M, 10M, 100M and 1B datapoints or table rows. The current architecture consists of MySQL database for storage and PHP backend, which does the computation of calculations. Thus, the first step is to research a method to remove this bottleneck from the application. This can be done by replacing one or several of the components with more scalable alternatives. First, I have decided to use Spark for the SQL queries from the database, which should improve the performance, as well as doing the computations with Spark. Secondly, I will integrate a data storage that is even more compatible with Spark. An example of such stoage are Apache Hadoop, HDFS, or Apache Kafka. I have decided to use HDFS for this implementation. However, replacing already complete implementations can be challenging and time consuming, because there may be compatibility issues.

It is not certain what the next bottleneck will be, but perceiving the current implementation it could be said that the program, which calculates the new prices isn't utilizing parallelism to its fullest. I can try to improve this by implementing the program with Spark and using the machine learning library H2O. It is also possible to try some more precise algorithms to improve the predictions.

The actual instance is running inside Docker. The instance uses ready-made docker image with Spark, and it is run on dedicated server, with 1 terabyte of hard disk space. The server has 64 Gigabytes of RAM and the CPU is 3.20Ghz Intel(R) Core(TM) i7-6900K with 8 hyperthreading cores.

Setting up Spark

Determining suitable environment to run Spark is very crucial task. The setup should be able to scale without much effort. I'm using Docker containers, one for running Spark applications and one for databases. For Spark I am using version 2.3.1 and for MySQL version 5.7. The version for MySQL is not the newest, but the reason it is being used is because the Dynamic Pricing Engine is using the same version and I am interested to see how that version interacts with Spark. The container for Spark also contains Hadoop and HDFS, therefore it can be used for deploying HDFS as well.

As discussed before, Spark can be run in several modes (Standalone, Yarn, Mesos). I'm testing mainly locally, therefore, I use the standalone mode. When the Spark program is executed, 'spark-submit' is called with the execution parameters. With these parameters it is determined how many executors are assigned per node and how many CPU cores are assigned for each executor. The two far end methods are either assigning 1 executor per node that uses all of the executors in the node, or assigning one executor per core which leads having as much executors as there are cores in a node. The optimal setup is usually in between these two extremes.

Listing 3: Spark Standalone setup with spark-submit

```
spark-submit --class SparkProgramClass --conf spark.network.
    timeout=3600s --conf spark.rpc.message.maxSize=2046 --
    conf spark.executor.extrajavaoptions=-Xmx16384m --num-
    executors=2 --executor-cores=16 --executor-memory=60G --
    driver-memory=48G --master=spark://spark:7077 --driver-
    class-path ${DIR}/mysql-connector-java-8.0.12.jar spark-
    program.jar
```

In Listing 3. Spark is instructed to have 1 executor on each node, and the executors use all available cores and memory.

5.1 Data transfer optimization

5.1.1 Spark + MySQL

First task is to establish connection between Spark and MySQL. After that I can compare the overhead and time consumed in transferring the data. One option is to create a MySQL table directly from CSV file that already has the data.

Listing 4: MySQL code for creating a table and loading a CSV file into it

```
CREATE TABLE salereports ( GPA VARCHAR(255) NOT NULL,
    order_charged_date DATE NOT NULL, order_charged_timestamp
     NULL DEFAULT CURRENT_TIMESTAMP, financial_status VARCHAR
    (255) NOT NULL, device_model VARCHAR(255) NOT NULL,
    product_title VARCHAR(255) NOT NULL, product_id VARCHAR
    (255) NOT NULL, product_type VARCHAR(255) NOT NULL,
    sku_id VARCHAR(255) NOT NULL, currency_of_sale VARCHAR
    (255) NOT NULL, item_price FLOAT NOT NULL,
    taxes_collected FLOAT NOT NULL, charged_amount FLOAT NOT
    NULL, city_of_buyer VARCHAR(255), state_of_buyer VARCHAR
    (255), postal_code_of_buyer VARCHAR(255),
    country_of_buyer VARCHAR(255) NOT NULL, PRIMARY KEY (GPA)
     );
```

```
LOAD DATA LOCAL INFILE '/path/to/file.csv' INTO TABLE
    salereports FIELDS TERMINATED BY ',' LINES TERMINATED BY
    '\n' IGNORE 1 ROWS (GPA, order_charged_date, @var1,
    financial_status, device_model, product_title, product_id
    , product_type, sku_id, currency_of_sale, item_price,
    taxes_collected, charged_amount, city_of_buyer,
    state_of_buyer, postal_code_of_buyer, country_of_buyer)
    SET order_charged_timestamp = FROM_UNIXTIME(@var1);
```

However, in order to have meaningful results, I need to have large enough datasets, from 1000 rows to 1 billion rows. This is because when implementing a big data solution, it is essential to test it with big data. Unfortunately, the actual data from the Dynamic Pricing Engine is in the low thousands. Therefore, I have generate the data myself and I have to make it be similar to the actual data. My first approach is to use MySQL for this. The solution is a query that generates random values for columns that are relevant, such as price and date, and inserts them into earlier created table. Another approach is using Spark, to essentially do the same thing, but instead of separate inserts, using the bulk insert option.

It is clear, that MySQL is not very efficient for this kind of data generation. This is because MySQL-inserts are able to use only approximately 10% of the disk lane. Adding more powerful hardware it is possible to create about 20 million rows with MySQL without crashing, but it still takes too long to be considered useful. MySQL uses only one core when running the SQL script, but the server I will be running it on has 48 cores. Using another script, it is possible to run the script on each core, thus making it possible to generate billion row datatable in approximately same time as 20 million rows. Therefore, at least in theory, the data generation could be done with MySQL alone.

However, a better option is to use Spark for the data generation and then write the created DataFrame to MySQL. Creating a DataFrame with 10 million rows of randomized data with Spark and writing it to MySQL takes 6,5 minutes and 100 million rows takes approximately 1 hour, as seen in Table 1. However, generating 1 billion rows comes with a issue regarding the limitations of one machine. The amount of memory needed for 1 billion rows exceeds the amount of memory that the machine I am using has. Generating that much data often leads to the following error: "java.lang.OutOfMemoryError: Java heap space". This is due to the executors needing to process so much at one given time that it does not fit into memory and as it was mentioned before, Spark functions faster when the entire data can fit in memory. In order to overcome this issue, I loaded the dataset of 100 million rows into Spark and transformed each row into 10 similar rows with a new column to make them unique. Then I set the partitions to 10000 in order to reduce the amount of memory needed per given time. After that, I wrote the created DataFrame into MySql. By doing this the application didn't raise errors and a MySQL table of one billion rows was created. However, with a machine with 16 CPU cores and 64 gigabytes of RAM it still took 44.2 hours. This execution time can be reduced by

Time spent generating data and writing it to MySQL					
Platform	1K - 100K rows	1M rows	10M rows	100M rows	1B rows
MySQL	1 min 2 sec - 19 min 12 sec	3 min 11,21 sec	1 hour 39 min 19.59 sec	> 1 day	No results
Spark	< 30 sec	2 min 12 sec	6 min 30 sec	1.2 hours	44.2 hours

Table 1: Time spent generating data and writing it to MySQL

Generating data with Spark and writing it to HDFS				
100K rows	1M rows	10M rows	100M rows	1B rows
11 sec	14 sec	43 sec	8 min 12 sec	1 hour 44 min

Table 2: Generating data with Spark and writing it to HDFS

either scaling out by adding more workes that bring more processing power and memory to the Spark cluster, or scaling up, by adding more memory to the machine.

The next step is to fetch the data from MySQL into the Spark program. In Scala, this is done by creating a DataFrame with MySQL options and loading the dataset into it. In Python, I use pysparks SQL context created from the SparkContext and load the data from the MySQL database. Reading 1 million rows from MySQL to Spark and making simple transformations (e.g., filter) and actions (e.g., count) can be done in 1.921190 s in Scala and 2.464203 s in Python. The slight difference in execution time is most likely due to compilation that has to be done from Python to Scala.

5.1.2 Spark + HDFS

Fetching data from HDFS is fairly similar to fetching data from any other database. First, a HDFS instance must be up and running and the data stored in it. I used Spark to write a CSV file into HDFS. Unlike MySQL, defining schema is not required when working with HDFS, as HDFS works with files. When generating data with Spark, I've noticed that the execution time grows linearly, when the data fits into memory, as seen in Table 2. If the size of data exceeds the size of memory, Spark responds to this by spilling the overhead to disk. This slows down the execution, but it is required in order to complete the execution.

When working with HDFS, similar steps have to done as with MySQL: I have SQLContext created from SparkContext that is able to load the CSV formatted data with no extra steps. After the data is loaded it can be transformed to Spark DataFrame and then further actions and transformations can be done.

5.2 Algorithm optimization

5.2.1 Random Forest

Random forest can be described as an ensemble of decision trees. They can be used both for regression and classification problems and are fast to train and predict. In addition random forests can easily be implemented in parallel [50]. In this thesis, the random forest prototype is implemented with H2O machine learning library and Spark, using the Sparkling Water framework.

To begin with a Spark Context is created as well as SQLContext from it. Next, the H2OContext can be created from the previously creates Spark Context. Next step is to load the data to the program. The data can be fetched from CSV file or directly from database. The loaded data format is H2O DataFrame. To do Spark actions and transformation on the dataset it is required to transform it to Spark DataFrame. Aftewards the DataFrame is once again transformed to H2ODataFrame. Now, it is possible to split the dataset into training, validation and testing frames. Now, the random forest estimator is created, which then can be trained with the training DataFrame as well as validate using the validation DataFrame. Finally, the trained and validated estimator can be used to predict values for the testing DataFrame.

5.2.2 Spark + CUDA

One way to improve the performance even further would be doing some of the calculations on the GPU. I made one prototype that uses Spark to fetch and transform the data and CUDA with Python to do the calculations. Practically this is done by creating a SparkContext and with it the dataset is parallelized, i.e., a RDD is made from the data. Then each of the partitions of the RDD can be processed by a function that has been instructed to be executed on GPU. This declaration is done with Python's Numba and Cuda package by wrapping the function with either '@vectorize' or '@cuda.jit' wrapper. However, this implementation is not in the scope of this thesis.

6 Results

In this section, I present answers to the research question as well as analyze the results that were received by using the suggested techniques earlier in this thesis. The research question was how to scale an existing application to handle vast data volumes. For this research, I decided to focus on implementations on local machine, and find suitable ways to improve the time complexity without using cloud computing platforms. This is because I think comparing local implementations gives better insight and the changes can be observed more accurately.

The Random Forest implementation that was discussed in previous chapter is not included in these comparisons, because the main scope of this thesis is to find feasible solutions for the current implementation of finding the price elasticity of demand, instead of developing a new method for it. However, it can be said that the the Machine Learning algorithms used in this implementations are very efficient in handling Big Data. The source code for the prototype implementation can be found in the Appendix of this thesis.

There are three implementations that are going to be compared. The first implementation is the current implementation in the Dynamic Pricing Engine that consist of PHP and MySQL. The results gained from here are used as reference to the new solutions. In second implementation, I have utilized Spark to fetch the data and process it and then pass it to the PHP pipeline. In third implementation, I use Spark together with HDFS running on top of Hadoop, which is one recommended architecture for handling big data processing. These implementations are such that they require more manual configuration and programming, but they give better insight of how parallelism and distributed systems work altogether, in comparison to, for example, just handling everything on a cloud platform.

The steps that are done with each implementation are:

- Fetch the data from the data storage (MySQL or HDFS)
- Filter the transactions (rows) that are relevant
- Calculate the optimal price using the price elasticity of demand
- Return the result

All of the implementations use same data with same datapoints; 1'000, 10'000, 100'000, 1'000'000, 10'000'000, 100'000'000 and 1'000'000'000.

It can be seen in both Figure 13 and Figure 14, that PHP with MySQL graph stops at 10 million transactions. This is because the Dynamic Pricing Engine was not able to calculate the optimal price for 100 million rows. The execution of the program was ended after approximately 45 minutes. It can also be seen that the execution time with PHP backend is slower than with Spark already in the 10 million mark.

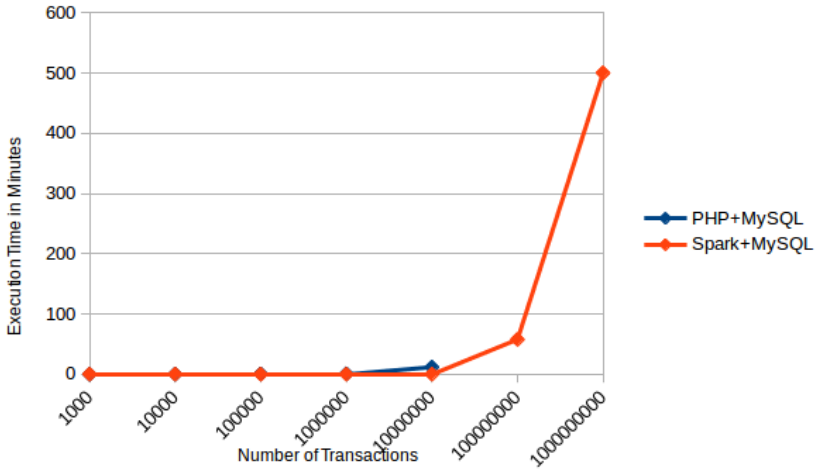

Figure 13: Comparison of Execution Times of PHP with Mysql and Spark with MySQL

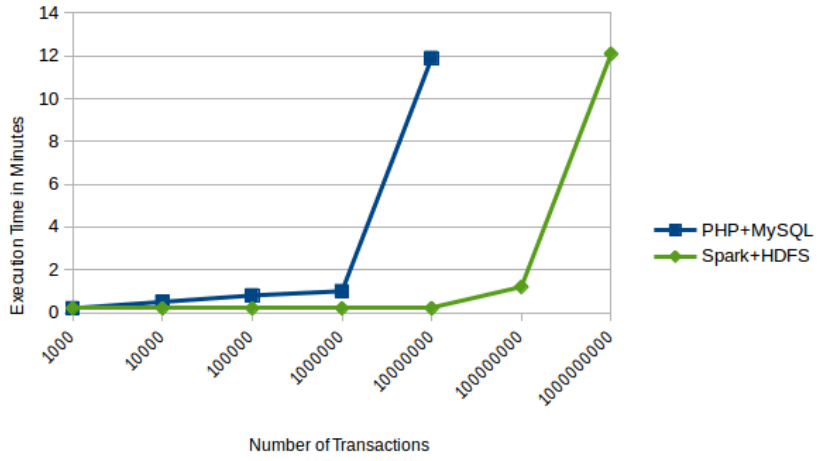

Figure 14: Comparison of Execution Times of PHP with MySQL and Spark with HDFS

In this comparison the data that Spark uses is stored in MySQL relational database.

Figure 15 shows the execution times for Spark utilizing data from HDFS and Spark using Mysql. As it can be seen, the growth of execution time of Spark with HDFS is almost linear. Spark with Mysql fares almost as well as Spark with HDFS when the amount of transaction is less 100 million, but after that there is clear difference in performance. It is important to notice, that the linear execution time for Spark with HDFS can only be achieved when the data fits in memory. Otherwise the data spills on disk, which slows down the execution.

In Figures 16 17 18 the execution times of all implementations are put together. From this it can be deduced, that using the recommended architecture of utilizing Spark with HDFS has the best performance. Spark utilizing HDFS was able to

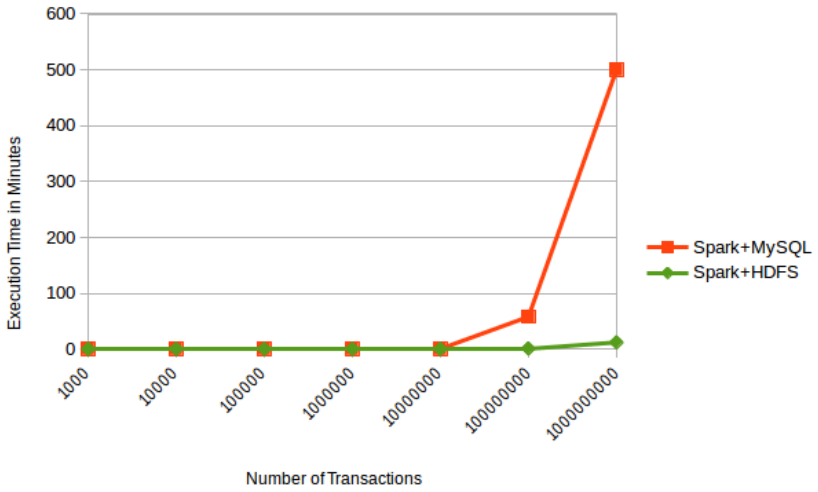

Figure 15: Comparison of Execution Times of Spark with Mysql and Spark with HDFS

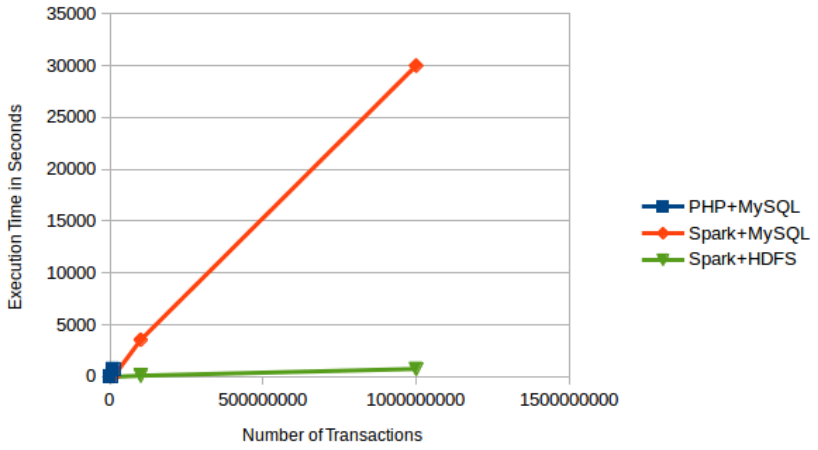

Figure 16: Comparison of Execution Times (seconds) of all the implementations

compute one billion rows in less than 15 minutes. For the Dynamic Pricing Engine these results mean, that by just doing the computations on Spark with the old MySQL database, while being able to complete the tasks, the scaling of the application is not as effective as desired. Thus, in order to make the application really into 'Big Data' application, the old architecure has to be replaced with even more scalable solution, such as replacing MySQL with HDFS. However, HDFS functions differently from SQL languages, which can lead to major problems in replacing the current, working solution. Other possible solution is to replace the current database with a cloud storage, which is provided by a cloud platform, such as S3 of AWS.

It was mentioned beforehand that the 'classic architecture's potential bottleneck is the database server, while faced with peak workloads'. This can be also seen in these

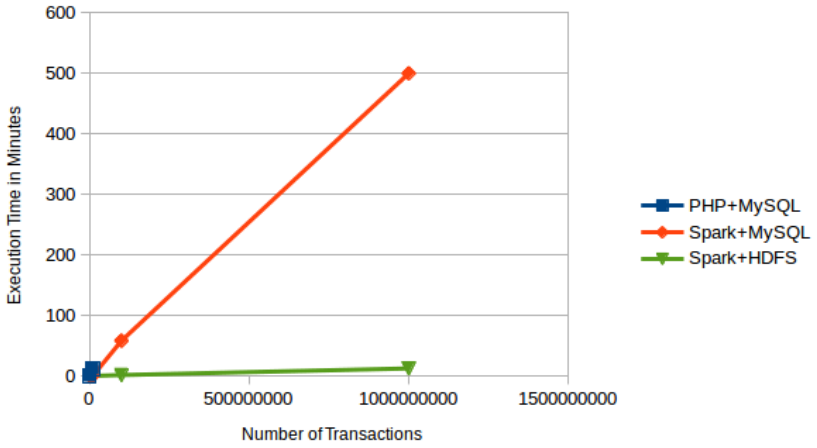

Figure 17: Comparison of Execution Times (minutes) of all the implementations

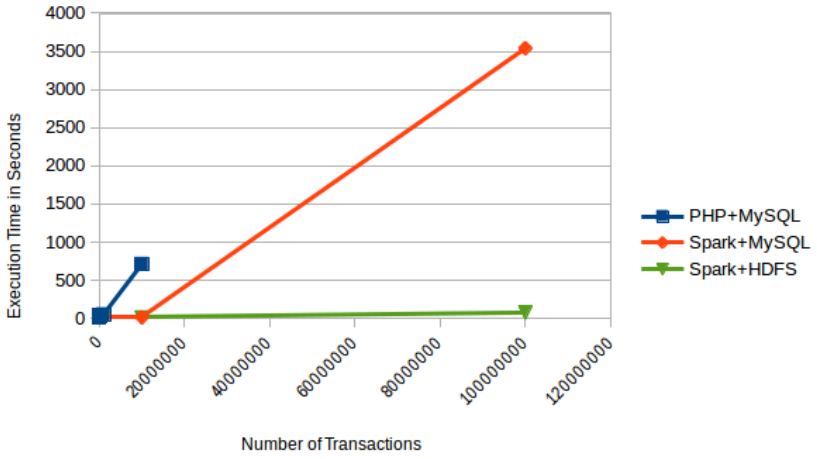

Figure 18: Comparison of Execution Times (seconds) of all the implementations from 1000 to 100 million rows

results, although the main problem is not with the peak workload, but rather with the capability of MySQL (with PHP or Spark) handling large datasets as efficient as with HDFS.

7 Summary

In this thesis, I looked into the problem of scaling up an existing system to accommodate data volumes of big data. Several propositions were made to increase efficiency and performance. The main focus was utilizing parallel techniques and distributing computing.

I covered the topics of big data, scalability and parallelism. I presented the basic concepts of distributed computing and tools to do it. Furthermore, the importance of big data and its analysis in gaming industry was discussed. I introduced the Dynamic Pricing Engine and explained the theory of price elasticity of demand, that the Dynamic Pricing Engine uses to solve the optimal prices for IAPs.

The research material and methods I selected and the reason for these choices were examined. In addition, the implementation of the prototypes was discussed along with the review of data generation. After that, I compared the results of these implementations, and I was able to show a significant difference based on whether the system uses parallel programming and distributed computation, or traditional programming models.

Based on the results suggestions were made for the new architecture, the major changes being replacement of the database into HDFS, and introducing Spark as the processing framework. The new architecture provides efficient tools for handling Big Data.

Working with big data has had a definite impact of my development insight. I think the term 'Big Data Thinking' is very descriptive of this new mindset. Basically this means that when implementing anything, one starts to consider the possibility of the application having to handle vast amount of data, thus driving the developer to at least think about the options to optimize the system in every stage.

I noticed several interesting issues, that are involved in processing and analyzing big data with parallel methods. Firstly, the amount of different approaches to program parallel applications was surprisingly large. Furthermore, many of these methods are programmer friendly by utilizing rich APIs to hide the complexity underneath. This seems to implicate that parallel processing is one of the best candidates for dealing with the big data problems that are sure to grow in number in the near future.

Second issue, that is important to figure out as soon as possible, is the testing of the implementation. When implementing a system that should be able to handle big data, it is required to test the implementation with equally complex data. The system can work like a charm with small data, but if it is not tested against big data, there is no guarantee it will handle it in the production. Another question rises when acquiring said test data: Where does it comes from? In order to have a proper dataset, it should be similar to the expected real data. In some cases, like in

this thesis, that kind of data can't be found, so it has to be self-generated. However, because the testing is usually first done in the local machine, it has to have enough storage capacity to accommodate the test data, which is now volumewise in par with big data. When the system is deployed, it can utilize scalable cloud storages or dedicated servers, but for the initial testing, the data resides in the local machine and can take up to terabytes of space.

Future Work

The next steps are full integration of the Spark application into the Dynamic Pricing Engine core, and further research on more developed prediction models for calculating the price elasticity of demand, such as Distributed Random Forest and Gradient Boosting. In addition, more research should be done on the cloud platforms, AWS, Azure and Google Cloud, to get more thorough analysis of their capabilities and pricing differences. One interesting platform is Google BigQuery, which has the storage solutions and processing power for very efficient big data handling. Furthermore, more research should be done on how GPU could be harnessed to do the calculations that are required.

References

[1] *Amazon EC2 Spot Instances official Website*. URL: https://aws.amazon.com/ec2/spot/.

[2] *Amazon Web Services official Website*. URL: https://aws.amazon.com/.

[3] Patrick L Anderson et al. "Price elasticity of demand". In: *McKinac Center for Public Policy. Accessed October* 13 (1997), p. 2010.

[4] *Apache Hadoop*. URL: http://hadoop.apache.org/.

[5] *Apache Hadoop MapReduce*. URL: https://hadoop.apache.org/docs/r3.1.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[6] *Apache HBase ™ Reference Guide*. URL: http://hbase.apache.org/book.html.

[7] *Apache Hive ™*. URL: https://hive.apache.org/.

[8] *Apache Pig*. URL: https://pig.apache.org/.

[9] *Apache Storm ™*. URL: http://storm.apache.org/.

[10] R. Arora. *Conquering Big Data with High Performance Computing*. Springer International Publishing, 2016. ISBN: 9783319337425. URL: https://books.google.fi/books?id=4H0XDQAAQBAJ.

[11] *AWS Fargate official Website*. URL: https://aws.amazon.com/fargate/.

[12] André B Bondi. "Characteristics of scalability and their impact on performance". In: *Proceedings of the 2nd international workshop on Software and performance*. ACM. 2000, pp. 195–203.

[13] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. "Graphics processing unit (GPU) programming strategies and trends in GPU computing". In: *Journal of Parallel and Distributed Computing* 73.1 (2013). Metaheuristics on GPUs, pp. 4–13. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731512000998.

[14] Charlie Catlett, Wolfgang Gentzsch, and Lucio Grandinetti. *Cloud computing and big data*. Vol. 23. IOS Press, 2013.

[15] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[16] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[17] *DeepAI, Neural Network*. URL: https://deepai.org/machine-learning-glossary-and-terms/neural-network.

[18] *Google Cloud*. 2018. URL: https://cloud.google.com/.

[19] WD Gropp et al. "Latency, bandwidth, and concurrent issue limitations in high-performance CFD". In: *Computational Fluid and Solid Mechanics* (2000), pp. 839–841.

[20] Developer Guide. *Amazon Elastic MapReduce*. 2010.

[21] Josef A. Habdank. *Extreme Apache Spark: how in 3 months we created a pipeline that can process 2.5 billion rows a day*. 2016. URL: https://www.slideshare.net/jozefhabdank/extreme-apache-spark-how-in-3-months-we-created-a-pipeline-that-can-process-25-billion-rows-a-day.

[22] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[23] Changqing Ji et al. "Big data processing in cloud computing environments". In: *Pervasive Systems, Algorithms and Networks (ISPAN), 2012 12th International Symposium on*. IEEE. 2012, pp. 17–23.

[24] Mariam Kiran et al. "Lambda Architecture for Cost-effective Batch and Speed Big Data Processing". In: *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*. BIG DATA '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 2785–2792. ISBN: 978-1-4799-9926-2. DOI: 10.1109/BigData.2015.7364082. URL: https://doi.org/10.1109/BigData.2015.7364082.

[25] Jacek Laskowski. *Mastering Apache Spark (2.3.1)*. URL: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/.

[26] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[27] *Microsoft Azure*. 2018. URL: https://azure.microsoft.com/.

[28] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[29] Vincent Natoli. *Why 2016 Is the Most Important Year in HPC in Over Two Decades*. URL: https://www.hpcwire.com/2016/08/23/2016-important-year-hpc-two-decades/.

[30] Frank Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, 2016.

[31] *NVIDIA Accelerated Computing Website*. URL: https://developer.nvidia.com/computeworks.

[32] *Official website of AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[33] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[34] *PyTorch GitHub Repository*. URL: https://github.com/pytorch/pytorch.

[35] Kevin Rands. *How big data is disrupting the gaming industry*. 2018. URL: https://www.cio.com/article/3251172/big-data/how-big-data-is-disrupting-the-gaming-industry.html.

[36] Jorge L Reyes-Ortiz, Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf". In: *Procedia Computer Science* 53 (2015), pp. 121–130.

[37] Eric Ries. *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books, 2011.

[38] Sandy Ryza et al. *Advanced Analytics with Spark : Patterns for Learning From Data at Scale*. Vol. First edition. O'Reilly Media, 2015. ISBN: 9781491912768. URL: http://search.ebscohost.com.libproxy.aalto.fi/login.aspx?direct=true&db=nlebk&AN=975341&site=ehost-live&authtype=sso&custid=ns192260.

[39] Eric Benjamin Seufert. *Freemium economics: Leveraging analytics and user segmentation to drive revenue*. Elsevier, 2013.

[40] Dilpreet Singh and Chandan K. Reddy. "A survey on platforms for big data analytics". In: *Journal of Big Data* 2(1).8 (2015). DOI: http://doi.org/10.1186/s40537-014-0008-6.

[41] Ion Stoica. *Apache Spark and Hadoop: Working Together*. 2014. URL: https://databricks.com/blog/2014/01/21/spark-and-hadoop.html.

[42] Marc A. Suchard et al. "Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures". In: *Journal of Computational and Graphical Statistics* 19.2 (2010). PMID: 20877443, pp. 419–438. DOI: 10.1198/jcgs.2010.10016. URL: https://doi.org/10.1198/jcgs.2010.10016.

[43] *TensorFlowOnSpark official GitHub Repository*. URL: https://github.com/yahoo/TensorFlowOnSpark.

[44] *The 5 Levels of Machine Learning Iteration*. URL: https://elitedatascience.com/machine-learning-iteration.

[45] M. Villamizar et al. "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. May 2016, pp. 179–182. DOI: 10.1109/CCGrid.2016.37.

[46] M Mitchell Waldrop. "More than moore". In: *Nature* 530.7589 (2016), pp. 144–148.

[47] Rui Xue. "SQL Engines for Big Data Analytics; SQL hakukone isoa datan analyysia varten". en. G2 Pro gradu, diplomityö. 2015, pp. 56+7. URL: http://urn.fi/URN:NBN:fi:aalto-201512165719.

[48] *Yahoo Developer Guide, MapReduce*. URL: https://developer.yahoo.com/hadoop/tutorial/module4.html.

[49] Lee Yang, Jun Shi, and Andy Chern Bobbie abd Feng. *Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters*. URL: http://yahoohadoop.tumblr.com/post/157196317141/open-sourcing-tensorflowonspark-distributed-deep.

[50] Cha Zhang and Yunqian Ma. *Ensemble machine learning: methods and applications*. Springer, 2012.

8 Appendix

A protoype of Distributed Random Forest implementation for predicting prices.

```
from pyspark.sql import Row, SparkSession, SQLContext
from pyspark.sql.types import *
from pyspark.sql import functions as F
from pyspark import SparkContext
from pysparkling import *
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import subprocess
from PIL import Image
import subprocess
import os

os.environ['PYSPARK_PYTHON'] = '/usr/bin/python3'

sc = SparkContext("local", "Pysparkling DRF")
sqlContext = SQLContext(sc)
hc = H2OContext.getOrCreate(sc)
import h2o
from h2o import *


def sale(SKU_id1, Currency_of_Sales, Item_Prices,
    Charged_Amounts, Country_of_Buyers):
     sale = Row(SKU_id1=SKU_id1,
                 Currency_of_Sales=Currency_of_Sales,
                 Item_Prices=Item_Prices,
                 Charged_Amounts=Charged_Amounts,
                 Country_of_Buyers=Country_of_Buyers)
     return sale

def refine_date_col(data, col):
     data["Day"] = data[col].day()
     data["Month"] = data[col].month()
     data["Year"] = data[col].year()
     data["WeekNum"] = data[col].week()
     data["WeekDay"] = data[col].dayOfWeek()
     data["HourOfDay"] = data[col].hour()

     data["Weekend"] = ((data["WeekDay"] == "Sun") | (data["
        WeekDay"] == "Sat"))
```

```
    data["Season"] = data["Month"].cut([0, 2, 5, 7, 10, 12],
        ["Winter", "Spring", "Summer", "Autumn", "Winter"])

f_sales = h2o.import_sql_table(connection_url="jdbc:mysql://
    localhost:3306/sales", table="salereports", username="
    username", password="password")

f_sales["item price"].table()

col_names = list(map(lambda s: s.replace(' ', '_'), f_sales.
    col_names))
f_sales.set_names(col_names)

refine_date_col(f_sales, "order_charged_timestamp")
f_sales.summary()

df_sales = hc.as_spark_frame(f_sales)
df_sales.createOrReplaceTempView("Sales")

salesHF = hc.as_h2o_frame(df_sales, framename="salesFrame")

salesHF.summary()

salesHF["sku_id"] = salesHF["sku_id"].asfactor()
salesHF["currency_of_sale"] = salesHF["currency_of_sale"].
    asfactor()
salesHF["country_of_buyer"] = salesHF["country_of_buyer"].
    asfactor()

salesHF.summary()

ratios = [0.6, 0.2]
frs = salesHF.split_frame(ratios, seed=12345)

train = frs[0]
train.frame_id = "Train"
valid = frs[2]
valid.frame_id = "Valid"
test = frs[1]
test.frame_id = "Test"

from h2o.estimators.random_forest import
    H2ORandomForestEstimator

predictors = salesHF.names[:]
```

```
response = "item_price"
predictors.remove("order_charged_date")
predictors.remove("order_charged_timestamp")
predictors.remove(response)


model_drf = H2ORandomForestEstimator(ntrees=50,
                                     max_depth=10,
                                     stopping_rounds=2,
                                     stopping_tolerance=
                                        0.01,
                                     score_each_iteration=
                                        True,
                                     max_hit_ratio_k=10,
                                     keep_cross_validation_predictions
                                        =True,
                                     keep_cross_validation_fold_assignmen
                                        =True,
                                     nfolds=3,
                                     fold_column="item_price
                                        ",
                                     seed=613374209)

model_drf.train(x=predictors, y="item_price", training_frame
   =train, validation_frame=valid)

image_file_name = "/path/to/file"
gv_file_path = "/path/to/gv_file"
model_file = model_drf.download_mojo(path="/path/to/tmp/
   mymodel", get_genmodel_jar=False)
h2o_jar_path = "/path/to/h20_jar"
mojo_full_path = model_file

def generateTree(h2o_jar_path, mojo_full_path, gv_file_path,
    image_file_path, tree_id = 0):
     image_file_path = image_file_path + "_" + str(tree_id)
        + ".png"
     result = subprocess.call(["java", "-cp", h2o_jar_path,
         "hex.genmodel.tools.PrintMojo", "--tree", str(
        tree_id), "-i", mojo_full_path, "-o", gv_file_path
        ], shell=False)
     result = subprocess.call(["ls", gv_file_path], shell =
        False)
```

```
def generateTreeImage(gv_file_path, image_file_path, tree_id
    ):
      image_file_path = image_file_path + "_" + str(tree_id)
         + ".png"
      result = subprocess.call(["dot", "-Tpng", gv_file_path,
          "-o", image_file_path], shell=False)
      result = subprocess.call(["ls", image_file_path], shell
          = False)

generateTree(h2o_jar_path, mojo_full_path, gv_file_path,
    image_file_name, 3)
generateTreeImage(gv_file_path, image_file_name,  3)

all_models = model_drf.cross_validation_models()
predictions = model_drf.predict(test)

(predictions['predict']==test['item_price']).as_data_frame(
    use_pandas=True).mean()

Img = Image.open('/path/to/tree_image')
Img.show()
```

Generating test data with MySQL

```
CREATE TABLE `test_data1m`
(
  `GPA`                          bigint(20) NOT NULL       AUTO_INCREMENT,
  `order_charged_timestamp`  timestamp  NULL DEFAULT CURRENT_TIMESTAMP,
  `financial_status`          VARCHAR(255)                 NOT NULL,
  `device_model`              VARCHAR(255)                 NOT NULL,
  `sku_id`                    VARCHAR(255)                 NOT NULL,
  `product_id`                VARCHAR(255)                 NOT NULL,
  `product_type`              VARCHAR(255)                 NOT NULL,
  `currency_of_sale`          VARCHAR(255)                 NOT NULL,
  `charged_amount`            FLOAT                        NOT NULL,
  `country_of_buyer`          VARCHAR(255)                 NOT NULL,
  PRIMARY KEY (`GPA`)
) ENGINE=InnoDB;


DELIMITER \$\$
CREATE PROCEDURE generate_data1m()
BEGIN
  DECLARE i INT DEFAULT 0;
  WHILE i < 1000000 DO
    INSERT INTO test_data1m(
        order_charged_timestamp,
        financial_status,
        device_model,
        sku_id,
        product_id,
        product_type,
        currency_of_sale,
        charged_amount,
        country_of_buyer
        )
    VALUES (
      FROM_UNIXTIME(
      UNIX_TIMESTAMP(
       '2018-01-01 01:00:00'
      )+FLOOR(RAND()*31536000)),
       'charged',
       'HWPRA-H',
      ELT(FLOOR(RAND()*12)+1,
         'product.a1',
         'product.a2',
         'product.b1',
         'product.b2' ,
```

```
    'z_product1.a1',
    'z_product1.a2',
    'z_product1.b1',
    'z_product1.b2',
    'z_product2.a1',
    'z_product2.b1',
    'z_product2.a2',
    'z_product2.b2'),
  'com.Producer.GameName',
  'inapp',
  ELT(FLOOR(RAND()*7)+1,
            'EUR',
            'JPY',
            'GBP',
            'KRW',
            'HKD',
            'SEK',
            'USD'),
  ELT(FLOOR(RAND()*20)+1,
            8.49,
            2.99,
            4.29,
            4.99,
            0.89,
            0.58,
            12.99,
            5.49,
            3.29,
            2.49,
            33.99,
            40.99,
            5.99,
            0.69,
            6.59,
            0.79,
            9.99,
            7.49,
            7.99,
            14.49),
  ELT(FLOOR(RAND()*8)+1,
                  'FI',
                  'JP',
                  'GB',
                  'KR',
                  'HK',
```

```
                        'NL',
                        'SE',
                        'US')
    );
    SET i = i + 1;
  END WHILE;
END\$\$
DELIMITER ;

CALL generate_data1m();
```

Spark implementation for creating 1 billion rows of data

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions.rand
import org.apache.spark.sql.functions.col
import org.apache.spark.sql.SaveMode
import org.apache.spark.sql.SQLContext

import java.sql.{Connection, DriverManager, SQLException}
import java.util.Properties
import java.util.Random

object GenerateData {

  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Generate
        Data").set("spark.sql.session.timeZone", "UTC")

    val randomTimestamp = udf((s: Int) => {
      s + scala.util.Random.nextInt(2000)
    })

    val title = Array(
      "product.a1",
    "product.a2",
    "product.b1",
    "product.b2",
    "z_product1.a1",
    "z_product1.a2",
    "z_product1.b1",
    "z_product1.b2",
    "z_product2.a1",
    "z_product2.b1",
    "z_product2.a2",
    "z_product2.b2")

    val currency = Array("EUR", "JPY", "GBP", "KRW" , "HKD", "SEK", "USD")
    val price = Array(
      8.49,
      2.99,
      4.29,
      4.99,
      0.89,
```

```
    0.58,
    12.99,
    5.49,
    3.29,
    2.49,
    33.99,
    40.99,
    5.99,
    0.69,
    6.59,
    0.79,
    9.99,
    7.49,
    7.99,
    14.49)

  val country = Array("FI","JP","GB","KR","HK", "SE","US")

  val rand = new Random(System.currentTimeMillis())
  val random_index_title = rand.nextInt(title.length)
  val result_title = title(random_index_title)

  val random_index_price = rand.nextInt(price.length)
  val result_price = price(random_index_price)

  val random_index_country = rand.nextInt(currency.length)
  val result_currency = currency(random_index_country)
  val result_country = country(random_index_country)

  val sc = new SparkContext(conf)

  Class.forName("com.mysql.jdbc.Driver")

  val sqlContext = new org.apache.spark.sql.SQLContext(sc)
  import sqlContext.implicits._

  // create Spark context with Spark configuration
  val connectionProperties = new Properties()
  connectionProperties.put("user", username)
  connectionProperties.put("password", password)
  connectionProperties.setProperty("Driver", "com.mysql.jdbc.Driver")

  val random_dataDF = sc.parallelize(
    Seq.fill(1000000000){(
      "charged",
      "HWPRA-H",
      title(rand.nextInt(title.length)),
      "com.Producer.GameName",
```

```
        "inapp",
        currency(rand.nextInt(currency.length)),
        price(rand.nextInt(price.length)),
        country(rand.nextInt(country.length))
      )}
    ).toDF(
      "financial_status",
      "device_model",
      "sku_id",
      "product_id",
      "product_type",
      "currency_of_sale",
      "charged_amount",
      "country_of_buyer")
    val random_data_with_timestampDF =
        random_dataDF.withColumn("order_charged_timestamp",
        randomTimestamp(lit(1516364153)))
    random_data_with_timestampDF.write.mode(SaveMode.Append)
            .jdbc("jdbc:mysql://localhost:3306/sales", "spark1b",
                connectionProperties)
  }
}
```