Management and Processing of Discographic Data with Amazon Elastic MapReduce

Pierluigi Videsott
University of Trento
Via Sommarive 9, 38123 - Povo (TN), Italy
pierluigi.videsott@studenti.unitn.it

Maurizio Astegher
University of Trento
Via Sommarive 9, 38123 - Povo (TN), Italy
maurizio.astegher@studenti.unitn.it

ABSTRACT

The purpose of this report is to explain how – by leveraging on the capabilities of the amazon web services – it is possible to manage and process a set of data that is too large and complex for traditional data processing techniques and technologies.
The report discusses the implementation of a set of services – from the retrieval of external data to its transformation, through the storage on non relational databases and finally the parallel computation on an external cluster – meant for the management of discographic information in order to easily join different data in an agile manner and subsequently perform additional processing based on the joined output.

Keywords

Discogs, Elastic MapReduce, Amazon, Amazon Web Services, SimpleDB, DynamoDB, S3, EMR, discographic, join, cluster, parallel, processing, storage

1. INTRODUCTION

With the increase in the amount of data that is available through the internet and the expectations that today's users have when interacting with web services, today's developers have to deal with the issues of managing and processing huge amounts of data in an amount of time that is acceptable by the average user.
In order to face such a cumbersome task, different solutions have been developed – and are currently being developed – in the field of distributed computing for the purpose of distributed storage and distributed processing. Such solutions allow developers to rely on cheap computer clusters built with commodity hardware thus empowering developed web services with relatively cheap, scalable computational power and highly available data stores.

For the purpose of this report – specifically the retrieval and parsing of large discographic datasets to be stored on a non-relational, highly available data store and subsequent computation of parallel joins using the MapReduce programming model – many different technologies have been taken into consideration.
Compared to other solutions, given the fact that it is well established in the big data space and with the additional benefit of having a consistent community of developers working with it, Hadoop has been chosen as the go to reliable, scalable, distributed computing framework for this project.
The next consequential choice that a developer faces is whether or not to install, configure and manage his own Hadoop cluster on one or more local machines, with the resulting parallelization limitations, or instead rely on external cloud computing platforms such as Amazon Web Services, Google Cloud or Microsoft Azure.
An additional option is also available which consists of downloading one of the QuickStart Virtual Machines made available by Cloudera which comes pre-installed with all the software that is needed for the purpose of this project thus bypassing all the issues related with the setup and configuration of the different tools required.

After an attentive evaluation of all the options, the choice fell on Amazon Web Services and more specifically on the Elastic MapReduce(EMR) service they offer. Arguably, it provides the best trade-off between ease of use and – by allowing computations to be performed on real clusters of multiple machines – suitability towards this report's goals.
Consequently, because of the ease in which other Amazon Web Services can be made to interact with EMR (considering also the availability of a free usage tier), the additional services and technologies needed have been selected from those included in the Amazon Web Services portfolio.

Following is a brief description of all the services and technologies that have been used throughout this project.

1.1 Discogs

Discogs is a website and database of information about audio recordings, which includes both commercial and off-label releases. It is especially known as the largest online database of electronic music releases, in particular on vinyl media, currently containing over 6 million releases by 4 million artists. The site's original goal was to build the most comprehensive database of electronic music, organized around the artists, labels, and releases available in that genre. Since then, it has expanded to include all the other genres.
In 2007, Discogs data became publicly accessible via a REST-

Project report, Department of Information Engineering and Computer Science, University of Trento, Italy. Course: Big data and social networks. Academic Year: 2015/16. Authors: Pierluigi Videsott (147562) and Maurizio Astegher (175195).

ful, XML-based API, but did not allow anyone to alter the data. On June 2011 version 2 of the API was released and the default response was changed from XML to JSON. Monthly data dumps are also provided in XML format.

1.2 Amazon Web Services

1.2.1 Amazon SimpleDB

Amazon SimpleDB is a highly available and flexible non-relational data store that offloads the work of database administration, so to let users focusing on application development without worrying about infrastructure provisioning, software maintenance, schema and index management, or performance tuning.
It automatically creates and manages multiple geographically distributed replicas of user data to enable high availability and data durability. As an application evolves, users can easily reflect these changes on the fly without worrying about breaking a rigid schema or needing to refractor code. It is also possible to choose between consistent or eventually consistent read requests, gaining the flexibility to match read performance (latency and throughput) and consistency requirements to the demands of the application.

1.2.2 Amazon S3

Amazon Simple Storage Service (Amazon S3), provides developers with secure, durable, highly-scalable object storage. Amazon S3 comes with a simple web interface to store and retrieve any amount of data from anywhere on the web. It provides cost-effective object storage for a wide variety of use cases including cloud applications, content distribution, backup and archiving, disaster recovery, and big data analytics.

1.2.3 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2 reduces the time required to obtain and boot new servernstances to minutes, allowing to quickly scale capacity as computing requirements change.

1.2.4 Amazon EMR

Amazon Elastic MapReduce (Amazon EMR) is a web service that makes it easy to quickly process vast amounts of data. It simplifies big data processing, providing an Hadoop framework running on the web-scale infrastructure of EC2 and Amazon S3. Users donâĂŹt need to worry about node provisioning, cluster setup, Hadoop configuration, or cluster tuning, and it is possible to provision one, hundreds, or thousands of compute instances to process data at any scale. Amazon EMR handles many big data use cases, including log analysis, web indexing, data warehousing, machine learning, financial analysis, scientific simulation, and bioinformatics.

2. OVERVIEW

Since the purpose of this project is to show how to manage a large amount of discographic data, the first step consists in finding a big dataset that provides such information. As discussed, Discogs is not only one of the biggest database of audio recordings currently available online but it also gives the possibility to access data without any charge. Access is provided either through their RESTful API (`https://www.discogs.com/developers/`) or monthly data dumps (which are continuously uploaded at `http://www.discogs.com/data/`).

The biggest limitation that a developer faces by using the API is that requests are throttled by the server to 20 per minute per IP address, making it impossible to download a vast amount of data in a reasonable time. Monthly data dumps instead provide all the information needed in just a few XML files. For the purpose of this project only the most recent information about artists (`discogs_20150601_artists.xml`) and releases (`discogs_20150601_releases.xml`) are meaningful, so other files will be ignored. The sizes of these two files, in their decompressed form, are respectively 755MB and 18.8GB.

Once downloaded, XML data needs to be cleaned and parsed multiple times before it can actually be used to perform any kind of operation involving the MapReduce framework. The data workflow is depicted in Figure 1: XML files are initially parsed in order to discard information which is either irrelevant or redundant. Cleaned results are converted into key-value form and continuously inserted into a Amazon SimpleDB domains.

Amazon Elastic MapReduce requires the input of MapReduce jobs to be stored inside the Amazon S3 Web Service; hence, data needs to be extracted from SimpleDB and cleaned again in order to be uploaded to S3. The easiest way to do this is to read tuple by tuple the content of the simpleDB domains and save the information in TXT file format. Amazon EMR can then be adopted to perform MapReduce operations on this input, provided that the source code of the job is stored as a JAR in Amazon S3, alongside the input files.

The first operation which is performed and discussed in this report is the natural join of the two original files which were downloaded from Discogs, namely artists and releases. The purpose of this operation is to associate to every artist their audio recordings, so to obtain a more exhaustive view of the data. The output of the MapReduce job is automatically stored into s3 in text format.
At this point, the joined view offers the possibility to perform more sophisticated operations such as counting the number of releases per artist to find out which is the one that has more albums to his name.

3. COMPUTING NATURAL JOIN

When processing large datasets the option of joining data by a common key can be very useful, if not essential. There is a straightforward way to join relations using MapReduce. Of the existing join patterns, reduce-side joins are the easiest to implement due to the fact that Hadoop sends identical keys to the same reducer. The `mapper` only pre-processes the tuples of the two datasets to organize them in terms of the join key. To perform the join, the `reducer` simply needs to cache a key from one of the datasets and compare it to the incoming keys of the other. As long as the keys match, we can join the values from the corresponding keys.

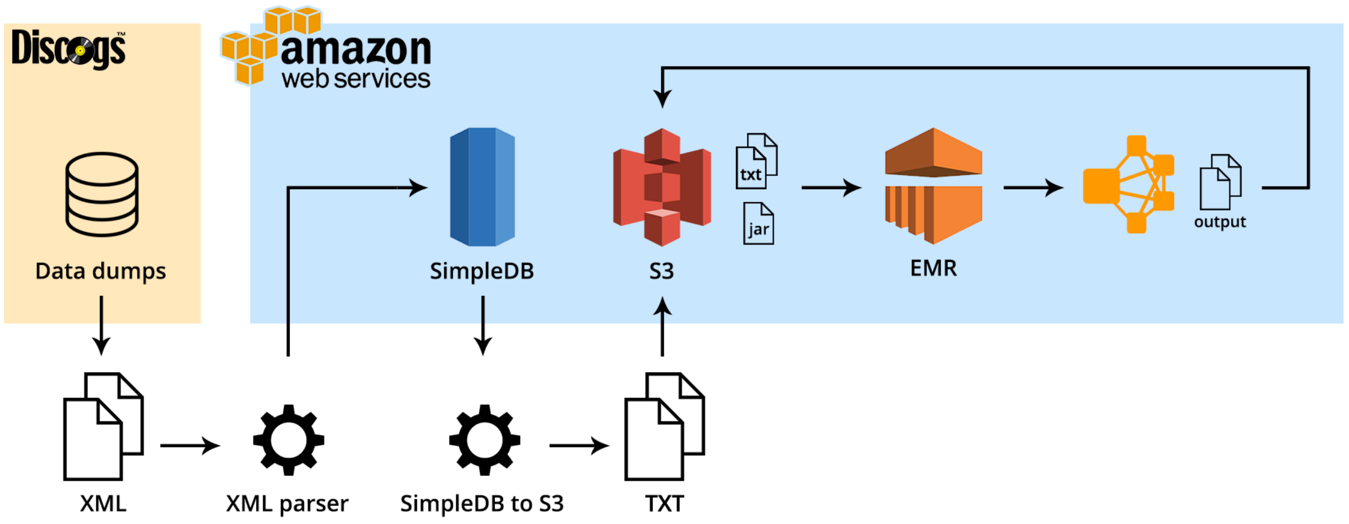


Figure 1: data workflow

More in detail, suppose relations R(A,B) and S(B,C) are each stored in a different file. To join these relations, we must associate each tuple from either relation with a key that is the value of its B-component.

A `map` process will turn each tuple (a,b) from R into a key-value pair with key b and value (a,R). Note that the relation is included with the value, so that, in the reduce phase, the matching is performed only on tuples from R with tuples from S, and not on pairs of tuples from R or pairs of tuples from S. Similarly, the `map` process turns each tuple (b,c) from S into a key-value pair with key b and value (c,S).
For each value b the `reduce` process will be associated with a list of pairs that are either of the form (a,R) or (c,S). The output from this key and value list is a sequence of key-value pairs, where The key is irrelevant. Each value is one of the triples (a,b,c) such that (a,R) and (c,S) are on the input list of values.

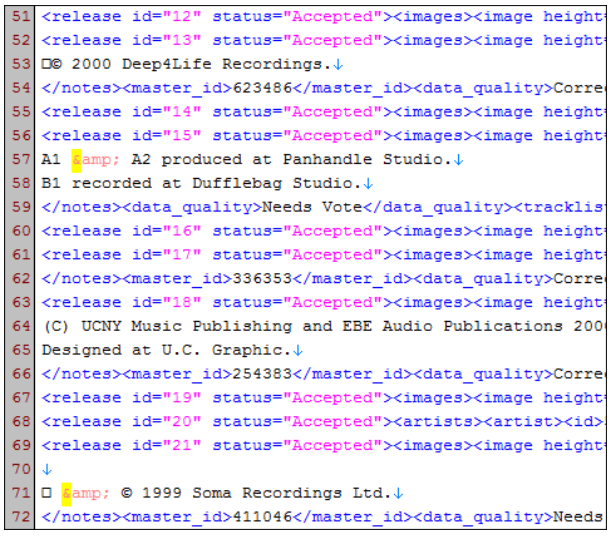
The same algorithm works if the relations have more than two attributes. Relation A can be seen as a representation of all the attributes in the schema of R but not S. B represents the attributes in both schemas, and C represents attributes only in the schema of S.

4. IMPLEMENTATION

A detailed explanation of how the project workflow works, how it evolved, together with an explanation of the most important issues and challenges encountered and related solutions that have been implemented, is provided next.

4.1 From Discogs to SimpleDB

As can be seen in Figure 1, the data workflow starts from `http://www.discogs.com/data/`. Discographic data, organized in the two XML file mentioned, are downloaded and cleaned locally with a SAX parser. A SAX parser works differently from a DOM parser, because it neither loads the XML document into memory nor it creates the object representation of the XML document; instead, the SAX parser uses callback functions to inform clients of the XML document structure. SAX parser are indeed essential to process the large amount of data that this project requires.



```
<release id="12" status="Accepted"><images><image height
<release id="13" status="Accepted"><images><image height
□© 2000 Deep4Life Recordings.↓
</notes><master_id>623486</master_id><data_quality>Corre
<release id="14" status="Accepted"><images><image height
<release id="15" status="Accepted"><images><image height
A1 &amp; A2 produced at Panhandle Studio.↓
B1 recorded at Dufflebag Studio.↓
</notes><data_quality>Needs Vote</data_quality><tracklis
<release id="16" status="Accepted"><images><image height
<release id="17" status="Accepted"><images><image height
</notes><master_id>336353</master_id><data_quality>Corre
<release id="18" status="Accepted"><images><image height
(C) UCNY Music Publishing and EBE Audio Publications 200
Designed at U.C. Graphic.↓
</notes><master_id>254383</master_id><data_quality>Corre
<release id="19" status="Accepted"><images><image height
<release id="20" status="Accepted"><artists><artist><id>
<release id="21" status="Accepted"><images><image height
↓
□ &amp; © 1999 Soma Recordings Ltd.↓
</notes><master_id>411046</master_id><data_quality>Needs
```

Figure 2: snippet of discogs_20150601_releases.xml

It is important to note that, even though it is of no issue for us to use the entire dataset, some data (such as the urls that point to the Discogs website's pictures) are discarded during the parsing because they are of little interest for the purpose of this report. For each artist the following information is maintained: the artist's name together with name variations and aliases, an artist identificator, a description, the list of groups to which the artist belongs and a set of urls to exernal information about the artist (such as artists' official websites). An artist can also represent a musical group, hence a list of members is associated to each record.
Each release contains an identifier, the release's title, the country of origin, the release date, the list of artists who contributed to the release, information about genres and sub-

genres, the complete tracklist, a list of official music videos, and also references to the company and the release's labels. Often the list of artists associated to each release presents duplicated information, but since the artists'id is used as the join keys for the two domains this could negatively effect the performance of the first MapReduce job (the one that joins artists with releases); hence, duplicates are removed from the lists of artists before they are inserted to SimpleDB.

All this information is fetched from the two XML files by executing the SAX parser, which creates a new item for each artist or release encountered; all these items are then inserted into two SimpleDB domains, one for the artists and the other for the releases. In order to increase the performance, write requests to SimpleDB are performed using batch requests, each of 25 generated items (the maximum allowed for a batch write request).

Initially, when deciding which storage technology was best for the project, the choice fell on DynamoDB. Since DynamoDB provides more powerful features than SimpleDB and is now considered the go to solution for most projects (so much so that SimpleDB is not advertised anymore as an Amazon Service and all the official tech support is outdated), choosing it over SimpleDB was the natural choice. During the initial development it quickly became clear that the very peculiar way in which DynamoDB bills its customers based on throughput (becoming very expensive when surpassing the 25 read/write requests per second), rather than on storage, not only limits the project's potential but also slows the achievement of its goals.
For those reasons the development rotated towards the older, hidden, less powerful and less documented SimpleDB which bills based on storage and doesn't place on the developer artificial throughput restrictions.

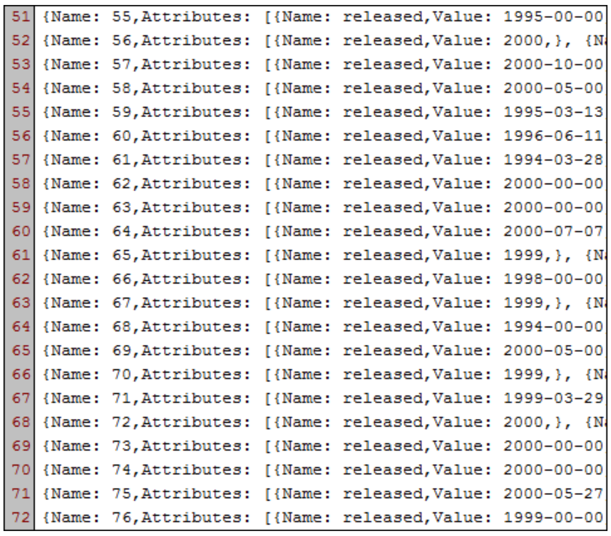
4.2 From SimpleDB to S3

Initially, for the sake of completeness, the objective was to have the MapReduce job work with input retrieved directly from and output written directly to SimpleDB. Despite great effort, it soon became clear that this option is not supported since Amazon does not provide developers with the libraries needed to do this. It is instead possible to perform MapReduce operations – from and to SimpleDB – by using Hive and its underlying query language which can indeed be used to perform MapReduce joins but it removes the developer's ability to implement and import custom Map and Reduce operations.

Therefore, in order to perform our first MapReduce job using the Amazon Elastic MapReduce framework, data has to be transferred from SimpleDB to Amazon S3. To do so, we establish a connection with SimpleDB querying the two tables; data is continuously parsed back into a string for each record and subsequently printed to files (one record for each row) in TXT format, again, one for the artists and one for the releases. The two TXT files are then uploaded to Amazon S3.

4.3 Elastic MapReduce

As hinted previously, in order to perform a customized MapReduce job on Amazon EMR, a JAR file – which contains the Java source code indicating how the job must be con-



```
{Name: 55,Attributes: [{Name: released,Value: 1995-00-00
{Name: 56,Attributes: [{Name: released,Value: 2000,}, {N
{Name: 57,Attributes: [{Name: released,Value: 2000-10-00
{Name: 58,Attributes: [{Name: released,Value: 2000-05-00
{Name: 59,Attributes: [{Name: released,Value: 1995-03-13
{Name: 60,Attributes: [{Name: released,Value: 1996-06-11
{Name: 61,Attributes: [{Name: released,Value: 1994-03-28
{Name: 62,Attributes: [{Name: released,Value: 2000-00-00
{Name: 63,Attributes: [{Name: released,Value: 2000-00-00
{Name: 64,Attributes: [{Name: released,Value: 2000-07-07
{Name: 65,Attributes: [{Name: released,Value: 1999,}, {N
{Name: 66,Attributes: [{Name: released,Value: 1998-00-00
{Name: 67,Attributes: [{Name: released,Value: 1999,}, {N
{Name: 68,Attributes: [{Name: released,Value: 1994-00-00
{Name: 69,Attributes: [{Name: released,Value: 2000-05-00
{Name: 70,Attributes: [{Name: released,Value: 1999,}, {N
{Name: 71,Attributes: [{Name: released,Value: 1999-03-29
{Name: 72,Attributes: [{Name: released,Value: 2000,}, {N
{Name: 73,Attributes: [{Name: released,Value: 2000-00-00
{Name: 74,Attributes: [{Name: released,Value: 2000-00-00
{Name: 75,Attributes: [{Name: released,Value: 2000-05-27
{Name: 76,Attributes: [{Name: released,Value: 1999-00-00
```

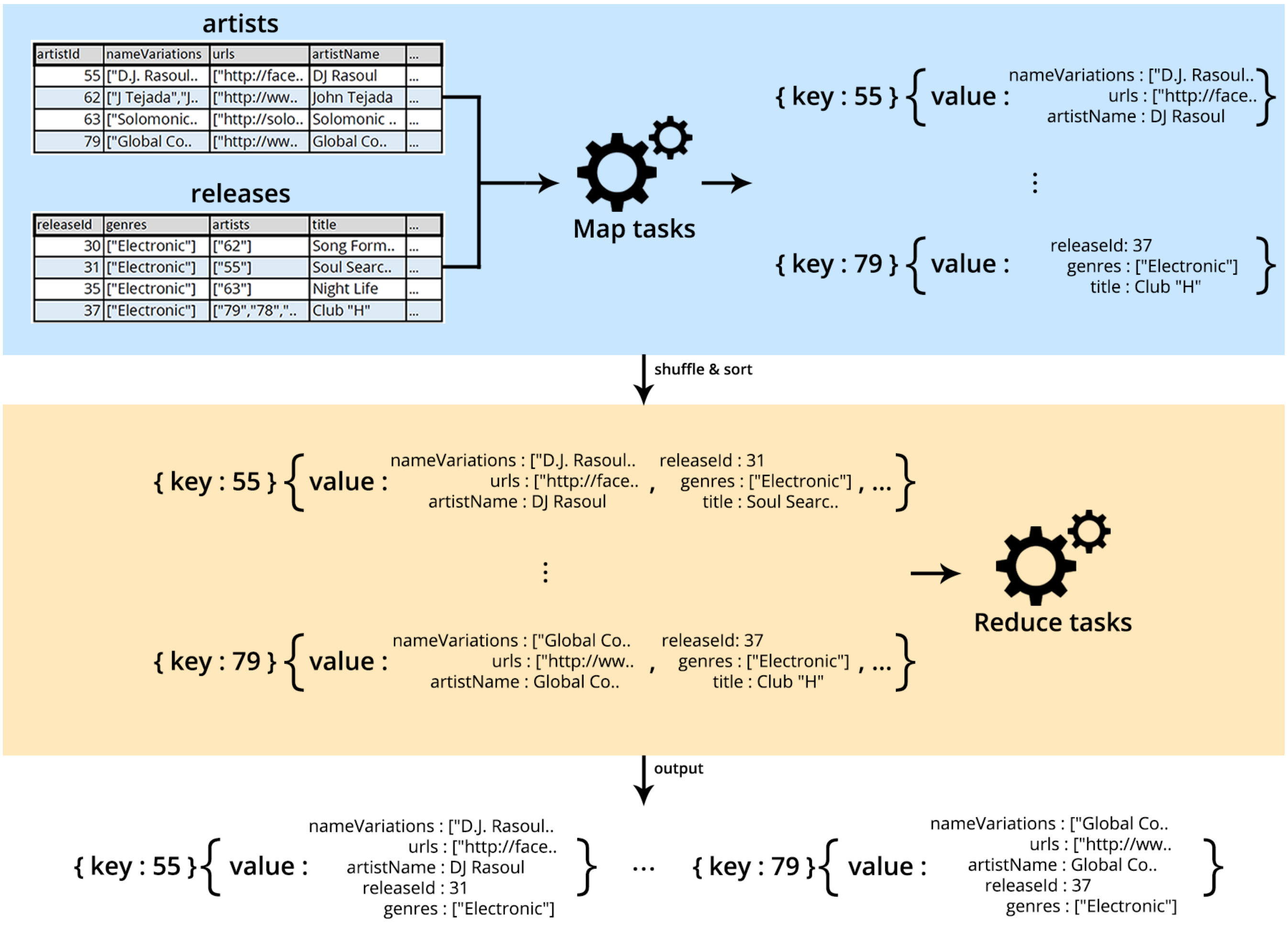
Figure 3: snippet of releases data as parsed from SimpleDB

figured and run – must be provided. The JAR contains the Main, the Map and the Reduce class. The Main class is responsible for specifying the format to be accepted as input and the format to be produced as output which, in this case, is set to text. Additionally, it is also responsible for specifying which class must be used as Mapper and which class as Reducer (in this case these are the Map and Reduce classes available in the JAR), and it also accepts two string inputs specifying the locations of the input files, which are used during the cluster setup, and the desired location for the output files.

The Map and Reduce classes are better explained with the aid of Figure 4 which shows a diagram of the implementation that has been chosen for this MapReduce join job. Highlighted in blue is the Map operation while in orange is the Reduce operation.

Once the ResourceManager of the Hadoop cluster has assumed the responsibility of distributing the software/ configuration to the slaves, schedule and monitor the tasks, and provide status and diagnostic information to the client, chunks of data from the artists and releases input files are distributed to the mappers. Representing these chunks, shown as tables in the diagram, are extracts from the artists and releases SimpleDB domains.

As said, each chunk's text line corresponds to a single SimpleDB tuple, so by analyzing them it is possible to understand the damain of origin. Tuples coming from the artists domain are turned by the mapper into key-value pairs, where the key is the artistId and the value is the union of all the other attributes. Tuples coming from the releases domain instead have to be turned into multiple key value-pairs, as many as the number of artists who contributed to the release. The artists attribute of this kind of tuples contains the list of contributing artists' identifiers; hence, each of them has to be taken by the mapper as the key of a new key-value pair, while the value is again the union of all the


artists
releases
Map tasks
{ key : 55 } { value : nameVariations : ["D.J. Rasoul.. urls : ["http://face.. artistName : DJ Rasoul }
⋮
{ key : 79 } { value : releaseId: 37 genres : ["Electronic"] title : Club "H" }
shuffle & sort
{ key : 55 } { value : nameVariations : ["D.J. Rasoul.. urls : ["http://face.. artistName : DJ Rasoul , releaseId : 31 genres : ["Electronic"] title : Soul Searc.. , ... }
⋮
{ key : 79 } { value : nameVariations : ["Global Co.. urls : ["http://ww.. artistName : Global Co.. , releaseId: 37 genres : ["Electronic"] title : Club "H" , ... }
Reduce tasks
output
{ key : 55 } { value : nameVariations : ["D.J. Rasoul.. urls : ["http://face.. artistName : DJ Rasoul releaseId : 31 genres : ["Electronic"] }
...
{ key : 79 } { value : nameVariations : ["Global Co.. urls : ["http://ww.. artistName : Global Co.. releaseId : 37 genres : ["Electronic"] }

artistId	nameVariations	urls	artistName	...
55	["D.J. Rasoul..	["http://face..	DJ Rasoul	...
62	["J Tejada","J..	["http://ww..	John Tejada	...
63	["Solomonic..	["http://solo..	Solomonic ..	...
79	["Global Co..	["http://ww..	Global Co..	...

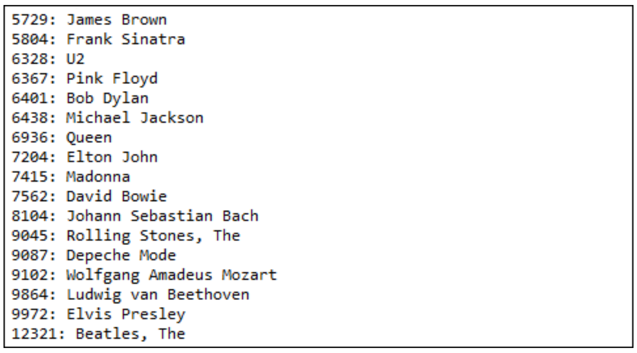
releaseId	genres	artists	title	...
30	["Electronic"]	["62"]	Song Form..	...
31	["Electronic"]	["55"]	Soul Searc..	...
35	["Electronic"]	["63"]	Night Life	...
37	["Electronic"]	["79","78","..	Club "H"	...

Figure 4: diagram of a reduce-side MapReduce join

other attributes' values.

Here ends the role of the mapper and the shuffle and sort operations performed by the framework come into play. Identical keys are sent to the same reducer meaning that each reducer will be assigned a list of values of both the artist's and the releases' attributes.
Since the matching is performed only on tuples coming from different domains, each reducer has to extract for each key the value that represents the artist, which is unique, matching it with all the other values given as input with the same key. The output from this key and value list is a sequence of key-value pairs, where the key is again the artistId and the value the combined information about artist and corresponding release. Again, an example of the output can be seen in Figure 4: for the artist with key 55 the output contains both the attributes of the artist and the attributes of one of the releases associated with that artist (releaseId 31).

Once the reduce-side join is finished, the output can easily be adopted as an input for another MapReduce job. In this case, similarly to the join, a MapReduce job that counts the number of releases for each artist has been implemented. Additionally – given the fact that the job of finding the highest number in a list cannot be parallelized – for the sake of completeness, an extra mapping is performed which naturally sorts the output by number of releases exploiting the shuffle and sort phase, thus returning an ordered list with the artist with the highest number of releases as the last artist.



```
5729: James Brown
5804: Frank Sinatra
6328: U2
6367: Pink Floyd
6401: Bob Dylan
6438: Michael Jackson
6936: Queen
7204: Elton John
7415: Madonna
7562: David Bowie
8104: Johann Sebastian Bach
9045: Rolling Stones, The
9087: Depeche Mode
9102: Wolfgang Amadeus Mozart
9864: Ludwig van Beethoven
9972: Elvis Presley
12321: Beatles, The
```

Figure 5: final output: artists with the biggest number of releases associated to them

An extra paragraph must be spent to highlight the issues that arose during the first tests performed on the entire dataset of both the artists and the releases. With such a big amount of data being processed, some unexpected garbage collection (GC) issues (GC overhead limit exceeded) caused the interruption of the job. This issue is the result of the GC trying to free memory but being pretty much unable to get anything done. By default it happens when the JVM spends more than 98% of the total time in GC and, when after GC, less than 2% of the heap is recovered. The solution adopted has been of rewriting the entire MapReduce job so as to reduce as much as possible the number of objects dynamically allocated in memory.

5. PERFORMANCE EVALUATION

When creating a cluster on Amazon EMR, during the setup, the following information must be provided:

- the directory location of the JAR file stored on S3;
- the Hadoop version to be used;
- the number of EC2 core instances;
- the EC2 instance type for both the master and the cores;
- the S3 directory location of the files to be used as input;
- the S3 directory location in which to save the output.

Out of these, the ones that have an impact on the computation's performance are the number of allocated EC2 instances, the instance type and the size of the input data. For all experiments, for both the master and the cores, the instance type has been set to m1.medium which is equipped with 1vCPU 3.75GB of memory and 410GB of storage. The number of allocated EC2 instances has instead been changed in different tests, in order to analyze the performance of the job on various input sizes. The input size of the entire artists dataset (1.13 GB) has not been reduced while the input size of the releases has been increased little by little.

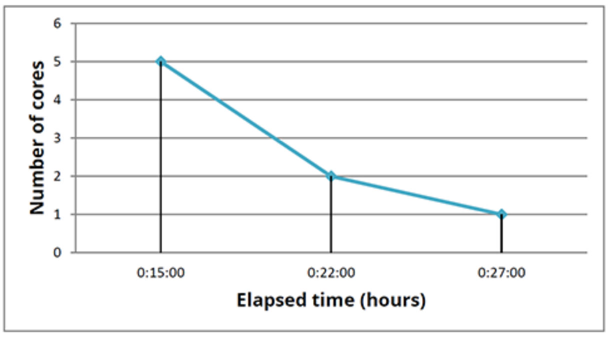


Figure 6: performance chart of 680MB releases dataset performed with 1, 2 and 5-cores cluster

Figure 6 shows the time needed to perform the join beetwen the artists and a small portion (680MB) of the releases; a cluster composed by only a master node and a single core node takes 27 minutes to complete the job, a cluster composed of 2 cores takes 22 minutes and one composed of 5 cores takes 15 minutes. It is important to note that all of these results (also those seen in Figure 7 and Figure 8) include an initial provisioning of the EC2 instances and configuration time of about 10 minutes, performed by Amazon at each cluster launch.

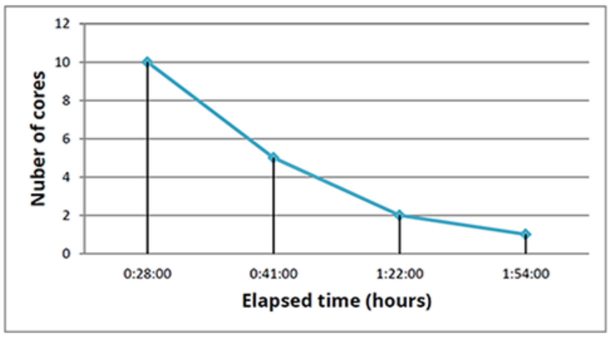


Figure 7: performance chart of 3.75GB releases dataset performed with 1, 2, 5 and 10-cores cluster

Figure 7 shows results obtained by a reduce-side join MapReduce job working with a releases input of 3.75GB. In this case, in addition to the other tests, also a 10-cores cluster has been adopted to asses performances.

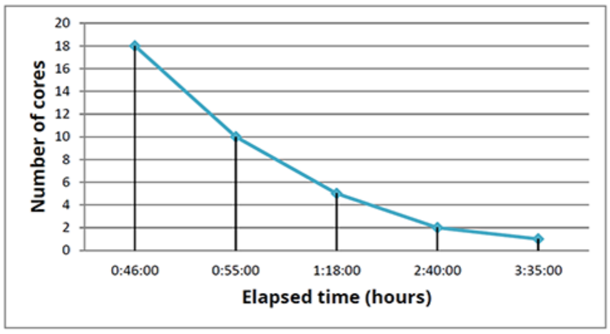


Figure 8: performance chart of 7.38GB releases dataset performed with 1, 2, 5, 10 and 18-cores cluster

Finally, the third chart (Figure 8) shows results obtained with a releases input of 7.38GB (the entire releases dataset) this time adding also a test case with a 18-cores cluster. The joined output reaches in this case a size of 36.7GB.

With this joined data it is now possible to perform the second MapReduce job, so to count the number of releases per artist. A 5-cores cluster has been adopted to test the performances of this operation, which ran for an hour.

In an ideal world, upgrading from a uniprocessor to an n-multiprocessor should provide about an n-fold increase in computational power. In practice this never happens because most of the computational problems cannot be effectively parallelized without incurring the costs of inter-processor communication ad coordination. As can be seen in the second and third charts (Figure 7 and Figure 8), specifically the difference between the tests performed with 10 cores and those performed with 5 cores, by doubling the computational power of the MapReduce jobs, the time elapsed does not halves.
This kind of analysis is very important for concurrent computation and it can be done using the Amdahl's Law, which captures the notion that the extent to which we can speed up any complex job is limited by how much of the job must

be executed sequentially.

Define the speedup S of a job to be the ratio between the time it takes one processor to complete the job versus the time it takes n concurrent processors to complete the same job; Amdahl's Law characterizes the maximum speedup S that can be achieved by n processors collaborating on an application, where p is the fraction of the job that can be executed in parallel. Assume that it takes normalized time 1 for a single processor to complete the job; with n concurrent processors, the parallel part takes time p/n and the sequential part takes time 1-p. Overall, the parallelized computation takes time:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

Therefore, the obtained results have overall been satisfactory and in accordance to Amdahl's Law.

6. CONCLUSIONS

The purpose of this report can be considered reached. It explains how, throughout the project, a very large discographic dataset has been retrieved and parsed in order to be stored in the non-relational, highly available data store SimpleDB. From SimpleDB, the data has been converted in a format that is uploadable to S3 and is accepted as input of the EMR join operation. Consequently, the output has been used to perform two additional MapReduce operations – though one is only used for sorting – with the goal of retrieving from the entire dataset the artist with the greatest number of records associated to him.

The entire project involved many different Amazon Web Services which are not included in the free tier. Luckily, upon request, Amazon provides students with 35€ to be spent on their services.
The total bill for the entire project amounts to 36.14€ with a total amount of computational instance hours – excluding learning, setup and failed tests – of 140.

Overall, the project has been of great interest and it proved to be a great learning experience.

7. REFERENCES

[1] Leskovec J., Rajaraman A., Ullman J. D., "Mining of Massive Datasets", chapter 2, 2014.

[2] Herlihy M., Shavit N., "The Art of Multiprocessor Programming", chapter 1, 2008.

[3] AWS SDK for Java Documentation. (2013). Retrieved from `http://aws.amazon.com/documentation/sdk-for-java/`

[4] Hadoop 2.4.1 Documentation. (08/04/2013). Retrieved from `http://hadoop.apache.org/docs/r2.4.1/`

[5] Joins with Map Reduce (26/02/2012). Retrieved from `https://chamibuddhika.wordpress.com/2012/02/26/joins-with-map-reduce/`

[6] MapReduce Algorithms - Understanding Data Joins. (26/06/2013). Retrieved from `http://codingjunkie.net/mapreduce- reduce-joins/`