Politecnico Di Milano

POLITECNICO
MILANO 1863

Nexmark Benchmarking Analysis using Apache Spark

Author

Mohamad Nezam <mohamad.nezam@mail.polimi.it>

Supervisor

Alessandro Margara <alessandro.margara@polimi.it>

Introduction

The Big Data world these days is expanding speedily and hugely. People are producing an enormous amount of data daily through many means, such as online shopping, communications, media consumption, etc.

Data, to be of value, needs to be operated on, sorted, and refined. So to do that, we need data processing systems such as Apache Kafka, Apache Spark, and Apache Flink. These processing systems, also called analytics engines, can operate on batch or streaming data.

However, Apache Spark is a cluster-computing framework for distributed programming that can operate on top HDFS and MapReduce, using a unified API. Apache Spark performs significantly faster than Hadoop MapReduce due to its in-memory processing. Possibly these are some of the reasons for its increasing popularity among big data developers. Also, Apache Spark ensures scalability and fault tolerance. In Apache Spark, any algorithm can run no matter how many processors are used (horizontal scalability with the data set is (vertical scalability). Fault tolerance stands for avoiding data in case a cluster node in the system fails.

Purpose

Studying and comparing different analytics engines gives us an insight into these data processing systems and when it is perfect to use a specific one among them, and the comparison between gains and losses of using each of them. Such awareness for any group of people requiring to operate and manage vast quantities of data would be so helpful for them because it would save them a lot of resources, time, and effort.

Moreover, the increasing demand to process the endless amount of incoming data was the main reason for implementing data process systems. Apache Spark is one such open-source platform that allows us to process events in both real-time and on-demand.

Nevertheless, some limitations and configurations can be modified to improve system efficiency and throughput.

Moreover, Apache Beam already did the benchmarking on Nexmark paper in different case scenarios for multiple data processing systems such as Spark, Flink, Google cloud, etc. We thought to help and contribute by adding results for other cases which are not yet done, such as running the nexmark queries using Apache Spark in Cluster mode.

saying that we reach to our main questions in this research :

1. What is the impact of core numbers on Spark Throughput?
2. What is the effect of the memory assigned to each executor on Spark Throughput?
3. What are the other parameters that can also leave an effect on the Spark Throughput?

We hope the reader will find answers to these questions after finishing reading this document. However, we will still keep searching for future work without stopping until we reach the most satisfying solutions for big data analysis users.

Related Work

Working with cloud-based frameworks such as Apache Spark to supercomputing frameworks is a topic of heightened interest in big performance distributed computing environments. We analyze related work on benchmarking and challenges of Spark development.

Benchmarking

O. Marcu and García-Gil present a comparison between Spark and Flink. Marcu provides a methodology that depends on a set of benchmarks (word count, tera sort, K-means, page rank, etc.) ran on up to one hundred nodes to understand the performance in this type of framework. García provides the results of three ML algorithms running on ten nodes, and each node has 16 cores. While, Our benchmark used AWS amazon instances, and it executed some of the nexmark paper queries but in a smaller cluster of 4 nodes and each node with eight cores.

Challenges of Apache Spark development

M. Armbrust gives feedback from a company deploying Apache Spark to different organizations. Their paper gives some of the main challenges faced, such as memory management or large-scale I/O. The authors worked on some memory management features as well as a customize network module. To let Apache Spark become way more accessible to non-experts, they also wrote an API that depends on data frames (like in R or Python).

R. Tous present an optimized deployment of Apache Spark onMareNostrum, the BSC's supercomputer. The authors generated a framework to automate Apache Spark usages. They also provide information on using Apache Spark, such as the number of workers, size of the workers.

Background

Data processing systems

Apache Hadoop and Apache Spark are the top two Big data technologies that have rapidly captured the IT market with various job roles available.
We will illustrate the restrictions of Hadoop for which Spark came into the picture and its drawbacks.

Apache Hadoop:

Apache Hadoop was built for batch processing. It takes extensive data set in the input, all at once, processes it, and provides the result. Batch processing is very effective in the processing of high-volume data. An output gets delayed due to the size of the data and the computational power of the system.

Its streaming engine uses Map-reduce, which is a batch-oriented processing tool. It takes extensive data set in the input, all at once, processes it, and produces the result.

MapReduce calculation data stream does not have any loops. It is a chain of steps. Each stage proceeds forward using an output of the previous step and producing input for the next scene. Furthermore, it embraced the batch-oriented model. The batch is processing data at rest. It takes a massive amount of data at once, processing it and then writing out the output.

Apache Hadoop supports batch processing only. It does not process streamed data. Therefore, performance is slower compared to Spark. Moreover, It provides configurable Memory management. We can do it dynamically or statically.

MapReduce is very fault-tolerant. Therefore, there is no necessity to restart the application from scratch in case of any malfunction in Hadoop. Furthermore, it has incredible scalability potential and has been used in production on tens of thousands of Nodes.

Tasks have to be manually optimized.
There are different ways to optimize the MapReduce tasks:

- Configure the cluster correctly
- use a combiner
- use LZO compression
- tune the number of MapReduce tasks appropriately
- use the most appropriate
- compact writable type for the data.

The MapReduce framework of Hadoop is almost slower since it is created to help the different formats, structures, and the massive volume of data. That is why Hadoop has higher latency than Spark. Moreover, it processes slower than Spark. The slowness occurs only due to the nature of the MapReduce-based performance. It produces lots of intermediate data, and much data is exchanged between nodes, thus creates immense disk IO latency. Moreover, it has to persist much data in disk for synchronization between phases to recover from failures. Additionally, there are no ways in MapReduce to cache all subsets of the data in memory.

Scheduler in Hadoop becomes the pluggable component.
There are two schedulers for multi-user workload:

1. Fair Scheduler
2. Capacity Scheduler.

To schedule complex flows, MapReduce needs an external job scheduler like Oozie. And, also it cannot cache the data inside the memory for future requirements.

Batch and Streaming processing

In Big Data and Data Analytics world, stream data processing and batch data processing are fundamental concepts. It is essential to know the distinction between each one of them. However, in batch processing, data is obtained first and then processed, whereas stream processing is in real-time, meaning information is transferred into the analytics tool piece-by-piece.

Batch processing

Batch processing frameworks are perfect for processing massive datasets that need much computation. Such datasets usually are bounded (limited set of data) saved on some persistent storage area.

Batch processing is proper for processing when the time is not sensitive to us, as processing a massive dataset would take some time.

Apache Hadoop's MapReduce is one of the most popular batch processing frameworks. MapReduce is a Java-based system for parallel processing for massive datasets. It divides the dataset into considerably smaller pieces and reads the data from the HDFS. Each part is then distributed and scheduled for processing between the available nodes in the cluster. Each node performs the needed processes on the chunk of data, and the intermediate results received are written back to the HDFS system. These intermediate results may then be redistributed, split, and assembled for further processing until final results are obtained and saved back to the HDFS system. The MapReduce programming model for processing data has two distinct tasks, a Reduce job, and a Map job.

Map job begins by taking a set of data and converting it into another set of data where individual data elements are broken into tuples consisting of key-value pairs. These key-value pairs may then be shuffled, sorted, and processed by one or more Map jobs.

The Reduce job takes the results of a Map job as its input and couples those data tuples into a minimal set of tuples.

Stream processing

If we require analytics results in real-time, then there is only one option which is stream processing. When the data is created, it is fed into the analytics tools using data streams. This helps us to receive nearly immediate results. Stream processing can be helpful in fraud detection because it will enable the real-time discovery of irregularities. Usually, in-stream processing is The latency is calculated in seconds or milliseconds. This is possible because in-stream processing data is analyzed before it hits the disk.

Examples of batch processing use cases include:

• Fraud detection

• Log monitoring

• Customer behavior analysis

• social media Analysing

Batch vs. stream processing

The kind of data that the scientist or data engineer is working with decides to a huge amount whether batch or stream processing is more optimal. Nevertheless, it is achievable to convert batch data into stream data to leverage real-time analytic results. This might allow the chance to react faster to opportunities or challenges in cases where time constraints apply.

Batch processing	Stream processing
Data is collected over a particular time	Data is collected continuously
Data is processed after it is all collected	Data is processed live, piece-by-piece
It can take an extended time and is more suitable for a large quantity of data with a low time restriction.	It is fast and more suitable for data that needs immediate processing.

Apache Spark

Since Apache Spark is an open-source cluster computing framework, it is widespread for real-time processing. Spark grants an interface for entire programming clusters with absolute fault tolerance and data parallelism. It builds on top of Hadoop MapReduce, extending the MapReduce model to use more types of computations efficiently. It's fundamentally used for analytics. The most significant reason people like Apache Spark is the speed. It is fast when used with substantial datasets.

Spark Features

Multiple language support (Polyglot): Apache Spark handles an RPC server to expose API from one language to another one. The source code is JVM objects wrappers. Moreover, an RPC server is running that exposes JVM objects (i.e., SparkContext) to external processes (PySpark and SparkR, etc.)**.** That helps us write in different languages such as java, scala, and python.

Speed: Spark is way faster than Hadoop MapReduce for massive-scale data processing. Spark can perform this speed by controlled partitioning. It handles data using partitions that help parallelize distributed data processing with minor network traffic.

Several factors make Apache Spark so fast:

1. In-memory Computation
Spark is meant to be for 64-bit computers that can handle Terabytes of data in RAM. Spark is designed in a way that it transforms data in memory and not in disk I/O. Hence, it cuts the reading/writing cycle processing time to disk and stores intermediate data in memory. This reduces processing time and the cost of memory at a time. Moreover, Spark supports parallel distributed data processing, almost 100 times faster in memory and ten times faster on disk.

2. Resilient Distributed Datasets (RDD)
The main abstraction of Apache Spark is Resilient Distributed Datasets (RDD). It is a fundamental data structure of Spark. Spark RDD can be viewed as an immutable distributed collection of objects. These objects can be cached using either a cache() or persist().

The beauty of storing RDD in memory using the cache() method is – while storing the value in memory if the data does not fit, it sends the excess data to disk or recalculates it. It is a logical partitioning of each dataset in RDD that can be computed on different cluster nodes. As it is stored in memory, RDD can be extracted whenever required without using the disks. It makes processing faster.

3. Ease of Use

Spark follows a general programming model. This model does not constrain programmers to design their applications into a bunch of maps and reduce operations. The parallel programs of Spark look very similar to sequential programs, which is easy to develop.

Finally, Spark works on a combination of batch, interactive, and streaming jobs in the same application. As a result, a Spark job can be up to 100 times faster and only need 2 to 10 times less code writing.

4. Ability for On-disk Data Sorting
Apache Spark is the largest open-source data processing project. It is fast when it stores a large scale of data on a disk. Spark has the world record of on-disk data sorting.

5. DAG Execution Engine
DAG or Direct Acrylic Graph allows the user to explore each data processing stage by expanding the detail of any stage. Through a DAG user can get a stage view that clearly shows the detailed view of RDDs.

6. SCALA in the backend
The core of Apache Spark is developed using the SCALA programming language, which is faster than JAVA. SCALA provides immutable collections rather than Threads in Java that helps in inbuilt concurrent execution. This is an expressive development APIs for faster performance.

7. Faster System Performance
Due to its cache property, Spark can store data in memory for further iterations. As a result, it enhances the system performance significantly. Spark utilizes Mesos, a distributed system kernel for caching the intermediate dataset once each iteration is finished.

Furthermore, Spark runs multiple iterations on the cached dataset,, and since this is in-memory caching, it reduces the I/O. Hence, the algorithms work faster and in a fault-tolerant way.

8. Spark MLlib
Spark provides a built-in library named MLlib, which contains machine learning algorithms. This helps in executing the programs in memory at a faster rate.

9. Pipeline Operation
Following Microsoft’s Dryad paper methodology, Spark utilizes its pipeline technology more innovatively. Unlike Hadoop’s MapReduce, Spark does not store the output fed of data in persistent storage. Instead, it just directly passes the output of an operation as an input of another operation. This significantly reduces the I/O operations time and cost, making the overall process faster.

10. JVM Approach
Spark can launch tasks faster using its executor JVM on each data processing node. This makes launching a task just a single millisecond rather than seconds. It just needs to make an

RPC and adding the Runnable to the thread pool. No jar loading, XML parsing, etc., are associated with it. Hence, the overall process is much faster.

11. Lazy Evaluation

Apache Spark holds its evaluation till it is required. This is one of the essential factors contributing to its high speed. For transformations, Spark adds them to a DAG (Directed Acyclic Graph) of computation, and solely when the driver requests some data does this DAGgets executed.

Multiple Formats: Spark supports various data sources such as Parquet, JSON, Hive, and Cassandra, apart from the typical formats such as text files, CSV, and RDBMS tables. The Data Source API gives a pluggable mechanism for accessing structured data though Spark SQL. Data sources can be more than just plain pipes that transform data and extract it into Spark.

Hadoop Integration: Apache Spark provides fluid compatibility with Hadoop. This is a benefit for all the Big Data engineers who started their careers with Hadoop. Spark is a possible replacement for the MapReduce functions of Hadoop, while Spark can operate on top of an existing Hadoop cluster using YARN for resource scheduling.

Spark Components

Spark main components are what make Apache Spark fast and dependable. Many of these Spark components were built to resolve the issues that cropped up while using Hadoop MapReduce. Apache Spark has the following components:

Spark Core: is the main engine for large-scale parallel and classified data processing. Further, additional libraries built on the top of the core grant diverse workloads for streaming, SQL, and machine learning. It is accountable for memory management and fault recovery, scheduling, classifying, and monitoring jobs on a cluster & interacting with storage systems.

Spark Streaming: the Apache Streaming module is a stream processing-based module within Apache Spark to process real-time streaming data. It uses the Spark cluster to provides the ability to scale to a high degree. Being based on Spark, it is also highly fault-tolerant, which means it can rerun failed tasks through checkpointing the data stream while it is being processed. Examples of streaming sources:

- TCP-based Stream Processing
- File Streams
- Flume Stream source
- Kafka Stream source

Spark SQL: is a modern module in Spark that combines relational processing with Spark's functional programming API. It supports querying data both via SQL and via the Hive Query Language. For those familiar with RDBMS, Spark SQL will be an easy shift from your earlier tools, where you can spread the limits of traditional relational data processing.

GraphX: is the Spark API used for graphs and graph-parallel computation. Thus, it increases the Spark RDD with a Resilient Distributed Property Graph. GraphX extends the Spark RDD abstraction at a high level by introducing the Resilient Distributed Property Graph (a directed multigraph with features appointed to each vertex and edge).

MLlib (Machine Learning): the Spark MLlib module equips machine learning functionality over several domains. However, there are several data types used (for example, vectors and the LabeledPoint structure). This module offers functionality that includes:

- Statistics
- Classification
- Regression
- Collaborative
- Filtering
- Clustering
- Dimensionality
- Reduction
- Feature Extraction
- Frequent Pattern Mining
- Optimization

Spark Main Parts

Spark Engine
Spark Engine schedules, distributes and monitors the data application across the spark clusters.

Spark Driver
Spark Driver is the program that operates on the master node of the machine and indicates transformations and developments on data RDDs. In simple terms, a driver in Spark formulates SparkContext, connected to a given Spark Master. The driver also carries the RDD graphs to Master, where the standalone cluster manager runs.

Spark Executor

When SparkContext is linked to a cluster manager, it obtains an Executor for the nodes within the cluster. Moreover, they are the processes that run computations and put the data on the worker node. The concluding tasks by SparkContext are carried to executors for their execution.

Spark Shared Variables
Spark has two types of shared variables: broadcast variables and accumulators.

- Broadcast variables enable the programmer to retain a read-only variable cached on every machine to provide every node a large input dataset copy efficiently.
- Accumulators are variables we only can add to them by commutative operation and an associative and can be efficiently supported in parallel. They defined to implement sums or counters.

Spark Architecture

The first thing to understand is clustering. A cluster is several machines, commonly referred to as nodes, working together as one unit. The advantage of clusters is to obtain more done in less time while not paying too much money. There are huge servers, like the IBM P-series, with many cores. Typically, they cost a lot more than several smaller machines with equal power combined. There is also a boundary to how much we can fill into one physical case, while clusters can scale over several devices thousands upon thousands if required.

Clusters can also be made to be more resilient. If our server breaks, everything stops. If several nodes disappear in a cluster, it will not affect much if we still have nodes remaining. The remaining ones will handle the workload. We should mention that while Apache Spark runs in a cluster, it technically does not have to be a multi-machine cluster, but Spark cannot stretch its legs on a single machine. Also, there is no reason to perform any overhead on an individual machine. Run any overhead on a single machine. The cluster notion is not new to Apache Spark and is not all that new within computer science. Like many other things in IT, it has been around since at least the 1960s. Historically, it's mainly been used for large-scale installations. High-speed networks and data amounts increasing faster than processing ability has put the technology in vogue for more use cases.

Figure 2.1 describes the Apache Spark installation architecture. An application is executed as many independent processes are distributed across a cluster, and SparkContext coordinates these processes. The said coordinator is an object found in the application's main() function, also known as the Driver Program. On top of that, SparkContext is connected to a Cluster Manager, and the Cluster Manager does the resource allocation.

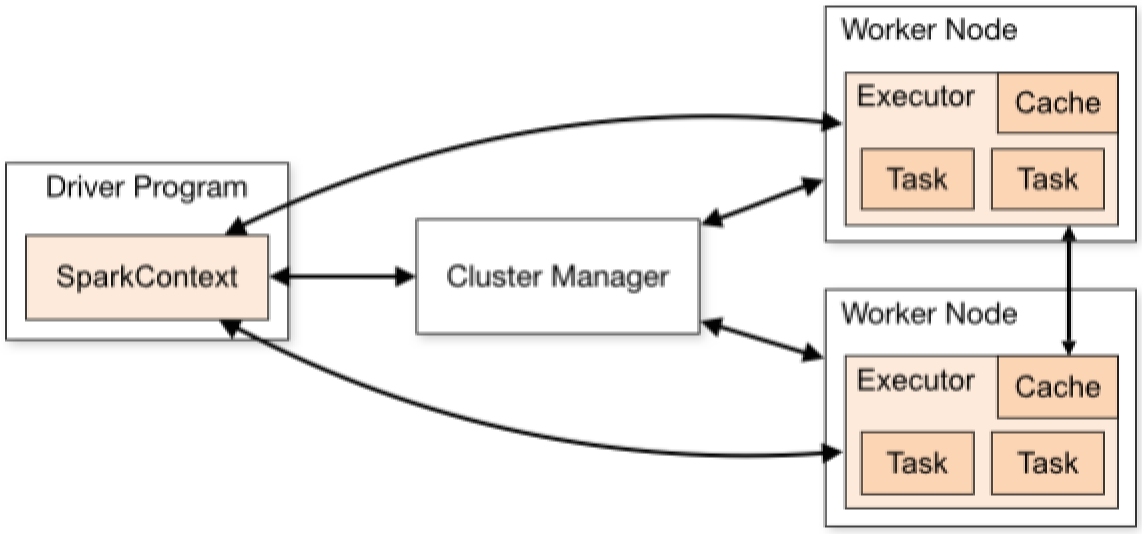


Figure 2.1: Architecture of Apache Spark in Cluster-Mode

Whenever a connection is verified, SparkContext obtains executors on the Worker Node instances. Each of these executors relates to only one application, which stores data and does computations. This indicates that applications running on the same cluster are executed in various JVMs SparkContext transmits the program to the executors as JAR or python files when the executors are acquired. Tasks are assigned to the executor processes, and every one of these processes can manage more than one task in various threads.

Apache Spark processing

Apache Spark is software that runs on top of the cluster. It handles the jobs for us. It looks at what requires to be performed, splits it into manageable tasks, and ensures it gets done. The core architecture is master/slave. The driver node is the controller and the head in the setup. This is where the status and state of everything in the cluster are kept. It is also from where you send your tasks for execution. When the job is being sent to the driver, a SparkContext is being initiated. This is the way our code communicates with Spark. The driver program then looks at the job, slices it up into tasks, creates a plan (called a Directed Acyclic Graph [DAG]), and figures out what resources it needs. The driver then checks with the cluster manager for these resources. When possible, the cluster manager starts up the necessary executor processes on the worker nodes. It tells the driver about them. The tasks are then being sent from the driver directly to the executors. They do what they are asked for and deliver the results. They communicate back to the driver so that they know the status of the entire job. Once all of the jobs are done, all the executors are freed up by the cluster manager. They will then be available for the next job to be handled. That is the whole flow and all components.

Working with data

Resilient Distributed Dataset (RDD) is the core data structure for Apache Spark. As the name indicates, the data is broadcasted over a cluster of machines. These parts are called partitions and include a subset of data. RDDs are also immutable, which means they cannot be modified, and they are resilient. So when modifications are made, new RDDs have to be generated. Commonly, there will be a lot of them in a job. If any of them are dropped, Spark can remake them by looking at how they were constructed in the first place. In RDD, the data is schemaless. It is excellent if we have unstructured data or our use case does not meet standard formats. Nevertheless, if we have organized information, there are other ways to work with data in Apache Spark: Datasets and DataFrames.

Storing data

There is also a storage layer to consider. A cluster needs data to be available on a shared file system. In Apache Spark, the Hadoop Distributed File System, HDFS, is being used.

workflow

The driver program is in the master node, which is driving the application. The code we are writing behaves as the driver program.

Inside the driver program, the first thing we do is creating a Spark Context, which we can assume is a gateway to all the Spark functionalities, which means anything we do on Spark goes through Spark Context.

The Spark Context works with the cluster manager to manage different jobs, and both Spark Context and Driver Program take care of the job execution within the cluster. A job is split into multiple tasks, which are distributed over the worker node. Anytime an RDD is created in Spark Context, it can be distributed across various nodes and cached there.

Worker nodes (slave nodes) job is to execute the tasks. These tasks are performed on RDDs that are partitioned in the worker node, and when it is finished, it returns the result to Spark Context. If we increase the number of workers, we can divide jobs into more partitions which means we can execute these jobs in parallel over multiple nodes, and sure this will be a lot faster. Also, by increasing the workers' number, memory size will also increase, which means we can cash the jobs to execute them more quickly.

We can divide the overall process into five steps:

- The client submits the spark application, then the driver converts user code into a logically directed acyclic graph (DAG).
- Then the driver converts the logical directed acyclic graph (DAG) into a physical execution plan with many stages, creates the tasks under each stage, and then sends them to the cluster.
- After that, the driver talks to the cluster manager to get the resources, so the cluster manager launches the executors in worker nodes and waits to receive the tasks from the

driver. When executors start, they register themselves with the driver so that the diver can have a complete view of it.
- The whole period the driver program will be monitoring all executors who are running, And the Drive node also will be scheduling the future tasks.

Monitoring spark streaming applications

We are generally interested in standard metrics such as the count of records in each micro-batch, the delay in running scheduled micro-batches, and how long each batch takes to run.

luckily Spark offers several monitoring interfaces that can help us, such as:

- **The Streaming UI:** A web interface that provides charts of key indicators about the running job
- **The Monitoring REST API:** A set of APIs that an external monitoring system can consume to obtain metrics through an HTTP interface
- **The Metrics Subsystem:** A pluggable Service Provider Interface (SPI) that allows for tight integration of external monitoring tools into Spark
- **The Internal Event Bus:** A pub/sub subsystem in Spark in which programmatic subscribers can receive events about different aspects of the application execution on the cluster.

However, we can also monitor and record application metrics from within the application by emitting logs. This requires running count().print(). Nevertheless, this may cause delays, adding to the application stages or performing unwanted shuffles that may be useful for testing but often prove expensive as a long-term solution.

Apache Spark listeners

Spark internally depend on SparkListeners for communication between its internal components in an event-based way. Also, the Spark scheduler emits events for SparkListeners whenever the stage of each task changes. SparkListeners listen to the events coming from Spark's DAGScheduler, which is the heart of the Spark execution engine. We can use custom Spark listeners to intercept SparkScheduler events to know when a task or stage starts and finishes.

The Spark Developer API provides eight SparkListener trait methods on different SparkEvents, mainly at start and stop, failure, completion, submission of receivers, batches, and output operation. We can execute an application logic at each event by implementing these methods.

SparkStreaming micro-batches

The default execution model in Spark Streaming is micro-batch. Spark starts a job in periods on an endless stream. Every micro-batch includes stages, and stages have tasks. Stages rely on the DAG and the action that the application code defines, and the number of tasks in each stage depends upon the number of DStream partitions.

At the origin of a streaming application, the receivers are committed to executing long-running tasks in a round-robin way.

Receivers build blocks of data rely on block-Interval. The BlockManager of the executors distributes the received blocks, and the network input tracker running on the driver is notified about the block places for moreover processing.

On the driver, an RDD is produced for the blocks in every batch-Interval. Every block transposes to a partition of the RDD, and a task is scheduled to process each partition.

Batch events

These events relate to the life cycle of batch processing, from submission to completion. Recall that each output operation registered on a DStream will lead to independent job execution. Those jobs are grouped in batches submitted together and in sequence to the Spark core engine for execution. This section of the listener interfaces fires events following the life cycle of submission and execution of batches.

As these events will fire following the processing of each micro-batch, their reporting rate is at least as frequent as the batch interval of the related StreamingContext. Following the same implementation pattern as the receiver callback interface, all batch-related events report a container class with a single BatchInfo member.

Each BatchInfo instance reported contains the relevant information corresponding to the reporting callback. BatchInfo also includes a Map of the output operations registered in this batch, represented by the OutputOperationInfo class. This class contains detailed information about each output operation time, duration, and eventual errors.

We could use this data point to split the total execution time of a batch into the time taken by the different operations that lead to individual job execution on the Spark core engine:

- **onBatchSubmited**

  It is called when a batch of jobs is submitted to Spark for processing. The corresponding BatchInfo object, reported by the StreamingListenerBatch Submitted, contains the timestamp of the batch submission time. At this point, the optional values processing-Start-Time and processing-End-Time are set to None because those values are unknown at this stage of the batch-processing cycle.

- **onBatchStarted**
- **onBatchCompleted**

  It is called when the processing of a batch has been completed. The provided BatchInfo instance will be fully populated with the overall timing of the batch. The map of OutputOperationInfo will also contain the detailed timing information for the execution of each output operation.

Benchmarking

introduction

Here in this thesis, we plan to evaluate the throughput of Nexmark research paper queries over a distributed cluster using Amazon Ec2 Instance.

In the beginning, the primary object was to contribute to the main project of the apache beam and start testing for spark streaming. However, after we finished reading and analyzing the GitHub code, it would not be possible to reuse the same code for our purpose for multiple reasons.

The main one was that the whole project code was in beam language, which still does not support Spark structured and only supports stream mode for testing purposes and is not recommended as it mentions on their official website. And the second one, the project code, was so difficult to break to extract a specific part to reuse again in a new project since it's a massive project and each piece connects with multiple components. However, to do that would take a lot of time, and we got only a small benefit from it, so that we started to do the coding from scratch.

System Design

Architecture

There are several different types of clusters managers where a spark app can allocate and deallocate the available physical resources such as memory, CPU, etc.

- **Standalone Cluster Manager**

  This is an effortless method for executing an app on a cluster, and it contains one Master and several Workers, each with configured CPU cores and memory size.

- **Hadoop yarn**

  This is another option for Cluster Manager in Spark. It supports utilizing varied data processing frameworks on the distributed resource pool. It is placed on the same nodes as Hadoop's Distributed File System(HDFS), which allows Spark to obtain HDFS data swiftly.

- **Apache Mesos**

  This one can perform both long-running utilities and analytics workloads on a cluster.

However, we used Standalone Cluster Manager in our experiments with one node as a Master and three nodes as Workers. Each of these nodes had 32G Ram and 8 CPU cores. In addition, to run the app in stream mode, we had another node with the exact specification to run the Kafka server and produce the event.

Benchmark dataset and workload configuration

We did the benchmark for both cases in batch and streaming mode using pre-generated events around (3,900,000) in total through generating them from the Apache Beam project and storing them in a text file.

We did that because the Apache Beam method generates events and storing them in the pipeline. Then the selected analysis engine would start the processing. And such a pipeline cannot be used as a feed source for spark input unless we wrote in beam language, which we mentioned before was not an option.

Since Spark can simply read the event directly from the text file, no particular configuration was needed in batch mode. In-streaming mode, we set up an extra node as a Kafka server to read the events from the text file and stream them to the Spark application.

Queries

In our use case, we used some of Nexmark research paper's queries which are :

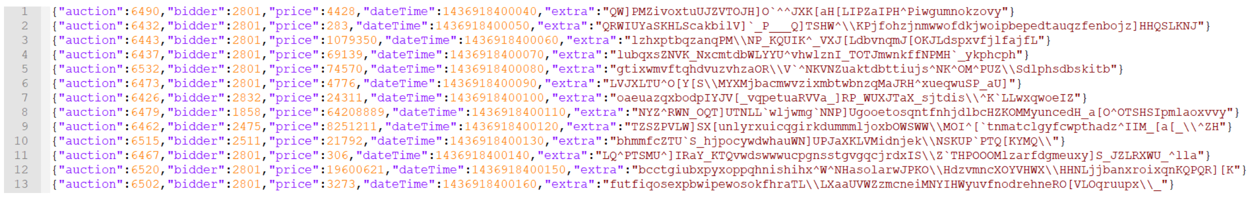
- **Query0** or **PASSTHROUGH**: Pass-through. Allows us to measure the monitoring overhead.
- **Query1** or **CURRENCY_CONVERSION**: What are the bid values in euros? Illustrates a simple map.
- **Query2** or **SELECTION**: What are the auctions with particular auction numbers? Illustrates a simple filter.

To help us in demonstrating our experiment.

Code Implementation

batch mode:

- As we mentioned, the first step was reading the events from the text file. In figure 3.1, we can see a sample of our input in JSON format stored in the text file.



```
{"auction":6490,"bidder":2801,"price":4428,"dateTime":1436918400040,"extra":"QW]PMZivoxtuUJZVTOJH]O`^^JXK[aH[LIPZaIPH^Piwgumnokzovy"}
{"auction":6432,"bidder":2801,"price":283,"dateTime":1436918400050,"extra":"QRWIUYaSKHLScakbilV]`_P___Q]TSHW^\\KPjfohzjnmwwofdkjwoipbepedtauqzfenbojz]HHQSLKNJ"}
{"auction":6443,"bidder":2801,"price":1079350,"dateTime":1436918400060,"extra":"lzhxptbqzanqPM\\NP_KQUIK^_VXJ[LdbvnqmJ[OKJLdspxvfjlfajfL"}
{"auction":6437,"bidder":2801,"price":69139,"dateTime":1436918400070,"extra":"lubqxsZNVK_NxcmtdbWLYYU^vhwlznI_TOTJmwnkffNPMH`_ykphcph"}
{"auction":6532,"bidder":2801,"price":74570,"dateTime":1436918400080,"extra":"gtixwmvftqhdvuzvhzaOR\\V`^NKVNZuaktdbttiujs^NK^OM^PUZ\\Sdlphsdbskitb"}
{"auction":6473,"bidder":2801,"price":4776,"dateTime":1436918400090,"extra":"LVJXLTU^O[Y[S\\MYXMjbacmwvzixmbtwbnzqMaJRH^xueqwuSP_aU]"}
{"auction":6426,"bidder":2832,"price":24311,"dateTime":1436918400100,"extra":"oaeuazqxbodpIYJV[_vqpetuaRVVa_]RP_WUXJTaX_sjtdis\\^K`LLwxqwoeIZ"}
{"auction":6479,"bidder":1858,"price":64208889,"dateTime":1436918400110,"extra":"NYZ^RWN_OQT]UTNLL`wljwmg`NNP]UgooetosqntfnhjdlbcHZKOMMyuncedH_a[O^OTSHSIpmlaoxvvy"}
{"auction":6462,"bidder":2475,"price":8251211,"dateTime":1436918400120,"extra":"TZSZPVLW]SX[unlyrxuicqgirkdummmljoxbOWSWW\\MOI^[`tnmatclgyfcwpthadz^IIM_[a[_\\^ZH"}
{"auction":6515,"bidder":2511,"price":21792,"dateTime":1436918400130,"extra":"bhmmfcZTU`S_hjpocywdwhauWN]UPJaXKLVMidnjek\\NSKUP`PTQ[KYMQ\\"}
{"auction":6467,"bidder":2801,"price":306,"dateTime":1436918400140,"extra":"LQ^PTSMU^]IRaY_KTQvwdswwwucpgnsstgvgqcjrdxIS\\Z`THPOOOMlzarfdgmeuxy]S_JZLRXWU_^lla"}
{"auction":6520,"bidder":2801,"price":19600621,"dateTime":1436918400150,"extra":"bcctgiubxpyxoppqhnishihx^W^NHasolarwJPKO\\HdzvmncXOYVHWX\\HHNLjjbanxroixqnKQPQR][K"}
{"auction":6502,"bidder":2801,"price":3273,"dateTime":1436918400160,"extra":"futfiqosexpbwipewosokfhraTL\\LXaaUVWZzmcneiMNYIHWyuvfnodrehneRO[VLOqruupx\\_"}
```

Figure 3.1: pre-generated events stored in the text file

- So first, we defined the text file's schema to save time since if we didn't, Spark would still be able to recognize it by himself, but that will take more time. After that, we cash it to help us reducing the processing time and increasing the throughput. Finally, we generate the view to start running the query we want.

```
val schema = new StructType()
      .add("auction", IntegerType)
      .add("bidder", IntegerType)
      .add("price", IntegerType)
      .add("dateTime", TimestampType)
      .add("extra", StringType)


val nexmark_df = spark.read.schema(schema).json("file:///2M.json")

val nexmark_df_Persist = nexmark_df.persist()

nexmark_df_Persist.createOrReplaceTempView("nexmarkEvents")
```

- The second step was executing the selected query. But before we start the executing, we get the starting time and when it finishes. Using this information, we can calculate the processing, time which is the time needed to execute the query. Hence we can calculate the throughput since we also know the total events number in the text.

```
val t1 = System.nanoTime

//-----------------------------------------------------------------
if(args(0)=="--query=0") {
      spark.sql("SELECT *  FROM nexmarkEvents").show()
      }
//-----------------------------------------------------------------

else if(args(0)=="--query=1") {
      spark.sql("SELECT auction,bidder,dateTime,price * 082  FROM
nexmarkEvents").show()

//-----------------------------------------------------------------

else if(args(0)=="--query=2") {
      spark.sql("SELECT auction,price FROM nexmarkEvents WHERE
auction = 1007 OR auction = 1020 OR auction = 2001 OR auction = 2019
OR auction = 2087").show()
    }

//-----------------------------------------------------------------

val t2 = System.nanoTime

val proccessingTime = (t2 - t1) / 1e9d

val Throughput = nexmark_df.count()/proccessingTime
```

- Finally, for analyzing purpose and speeding up running different experiments sequentially, every time the execution finishes, we store all related information we collected during the execution in a new line in the text file separated by a semicolon to help us later.

```
val fileName = "file:///batch.txt"
val file = new File(fileName)
val bw = new BufferedWriter(new FileWriter(file,true))
val text=args(0)+ ";" + args(1) + ";" +
     nexmark_df_Persist.rdd.getNumPartitions + ";" +
     readingTime + ";" + proccessingTime + ";" + totalTime + ";" +
     nexmark_df.count() + ";" + proccessingThroughput + ";" +
     totalTthroughput + "\n"

bw.write(text)
bw.close()
```

- And to save time and speed up the running of the experiments, we wrote a simple script that will run the experiment multiple times with different input arguments immediately when the previous one finishes.

```
for query in 0 1 2
do
  for coreNumber in seq(1 1 21)
  do
  spark-submit --master spark://172.31.45.209:7077 --total-executor-
cores $coreNumber  --executor-cores 1 --executor-memory 2g --driver-
memory 4g SparkKafkaBatch-assembly-0.3.jar --query=$query
cores{$coreNumber}executor-memory{2g}driver-memory{4g}
  sudo rm -r  /opt/spark/work/* ;
  done

  echo "Done $query  with $coreNumber core"
done
```

steam mode:

- For streaming mode, we used Apache Kafka to produce the events for us, where this time Apache Kafka read the event from the text file and sending to our Spark Application as a stream where our application is listening and waiting for these events.

- So first, our Spark app code is very similar to the code for batch mode with some modification, such as at the beginning, instead of reading the event from a text file now we configure it to read the event from Kafka sink where we specify the necessary setting for that such as batch interval, Kafka host server, Kafka topic, number of events to get before stopping the app.

```
val conf = new SparkConf().setAppName("nexmarkWithSpark")
val sc = new SparkContext(conf)
val r = scala.util.Random
val groupId = s"stream-checker-v${r.nextInt.toString}"
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> args(1),
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> groupId,
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)
val schema = new StructType()
  .add("auction", IntegerType)
  .add("bidder", IntegerType)
  .add("price", IntegerType)
  .add("dateTime", TimestampType)
  .add("extra", StringType)

val topics = Array(args(2))
val batchInterval = Seconds(args(3).toInt)
// How many events to get before terminating
val eventsToGet = args(4)
val ssc = new StreamingContext(sc, batchInterval)
val stream = KafkaUtils.createDirectStream[String, String](
  ssc,PreferConsistent,Subscribe[String, String]
 (topics, kafkaParams))

val batchNumber =
             ssc.sparkContext.longAccumulator("totalEventsCount")
val totalEventsCount =
             ssc.sparkContext.longAccumulator("totalEventsCount")

val messages = stream.map(record => record.value)
```

-
- And then, when the app starts, the received events will be stored in the messages object, which will be treated the same way we processed events in batch mode. The only difference is at the end of each batch interval, and we check the total incoming events so far if it reaches the specified number taken from the input so that we can stop the experiment automatically.

```
messages.foreachRDD { rdd =>
      // Now we want to turn RDD into DataFrame
      val spark =
SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()

      val rawDF = rdd.toDF("msg")
      val df = rawDF.select(from_json($"msg", schema) as
                            "data").select("data.*")
      df.cache()
      df.createOrReplaceTempView("nexmarkEvents")
      //------------------------------------------------------------
      if(args(0)=="--query=0") {
        spark.sql("SELECT *  FROM nexmarkEvents").show()
      }
      //------------------------------------------------------------
      else if(args(0)=="--query=1") {
        spark.sql("SELECT auction,bidder,dateTime,price * 082  FROM
        nexmarkEvents").show()
      }
      //------------------------------------------------------------
      else if(args(0)=="--query=2") {
        spark.sql("SELECT auction,price FROM nexmarkEvents WHERE
                   auction = 1007 OR auction = 1020 OR auction = 2001
                   OR auction = 2019 OR auction = 2087").show()
      }

      if (totalEventsCount.value >= eventsToGet.toInt ) {
          ssc.stop()
      }
    }
```

- And finally, we add a spark listener, which will be auto-triggered at the end of each batch to help calculate the throughput.

```
val listen = new
    JobListener(ssc,batchNumber,totalEventsCount,args(0),args(5))
ssc.addStreamingListener(listen)
ssc.start()
ssc.awaitTermination()
```

- We calculated the throughput at the end of each batch by getting the total incoming number of events during this batch and divided it by the processing time was taken to do so. Which as we mentioned before, the processing time has been calculated automatically for us through Spark Metrics at the end of each batch. After that, as usual, we wrote these results to a text file to analyze them at the end of the experiments.

```
override def onBatchCompleted(batchCompleted:
      StreamingListenerBatchCompleted) = synchronized {

      if(batchCompleted.batchInfo.numRecords!=0){
          batchNumber.add(1)
          val througput1 =
                  batchCompleted.batchInfo.numRecords/batchCompleted
                  .batchInfo.processingDelay.get

          totalEventsCount.add(batchCompleted.batchInfo.numRecords)
          val fileName = "data.txt"
          val file = new File(fileName)
          val bw = new BufferedWriter(new FileWriter(file,true))
          val text = {
                      args_0+";coreNumber;"+
                      args_6+";batchNumber;"+
                      batchNumber.value.toString()+";"+
                      batchCompleted.batchInfo.batchTime.toString()+
                      ";numRecords;"+
                      batchCompleted.batchInfo.numRecords.toString()+
                      ";processingDelay;"+
                      througput1.toString()+";totalEventsCount;"+
                      totalEventsCount.value.toString()+";"+ "\n"
        }
        bw.write(text)
        bw.close()
      }
    }
  }
```

Results

batch mode:

As we can observe from the following table, an example of the information stored in a text file during multiple experiments we did for query 0.

query	cores	read events time	RDD Partition number	processing time	throughput
0	1 core	2.807359578	5	7.880609986	495758.1973
0	2 core	2.827712468	5	8.15487115	479085.0681
0	3 core	2.807026337	5	8.272651859	472264.1623
0	4 core	2.815826107	5	7.907216221	494090.068
0	5 core	3.009960546	5	7.62152901	512610.6579
0	6 core	2.84917755	6	7.168753673	544986.9222

0	7 core	2.814154879	7	6.80480404	574135.1223
0	8 core	2.778502664	8	6.526277422	598637.8984
0	9 core	2.783271652	9	6.39200217	611213.3407
0	10 core	2.801802702	10	6.614766436	590629.622
0	11 core	2.799002952	11	6.061296904	644561.232
0	12 core	2.818839181	12	5.898065996	662399.6752
0	13 core	2.804178323	13	5.827696532	670398.1545
0	14 core	2.772236065	14	5.769163567	677199.9016
0	15 core	2.834262868	15	5.736474713	681058.8725
0	16 core	2.814205785	16	5.663075884	689886.0407
0	17 core	2.833821143	17	6.171601738	633041.0104
0	18 core	2.908753928	18	5.661766315	690045.6117
0	19 core	2.903745136	19	5.659106541	690369.9324
0	20 core	2.803935767	20	5.69405533	686132.6021
0	21 core	2.804498244	21	5.395298482	724126.2023

And the following charts explain the throughput calculated for each query:

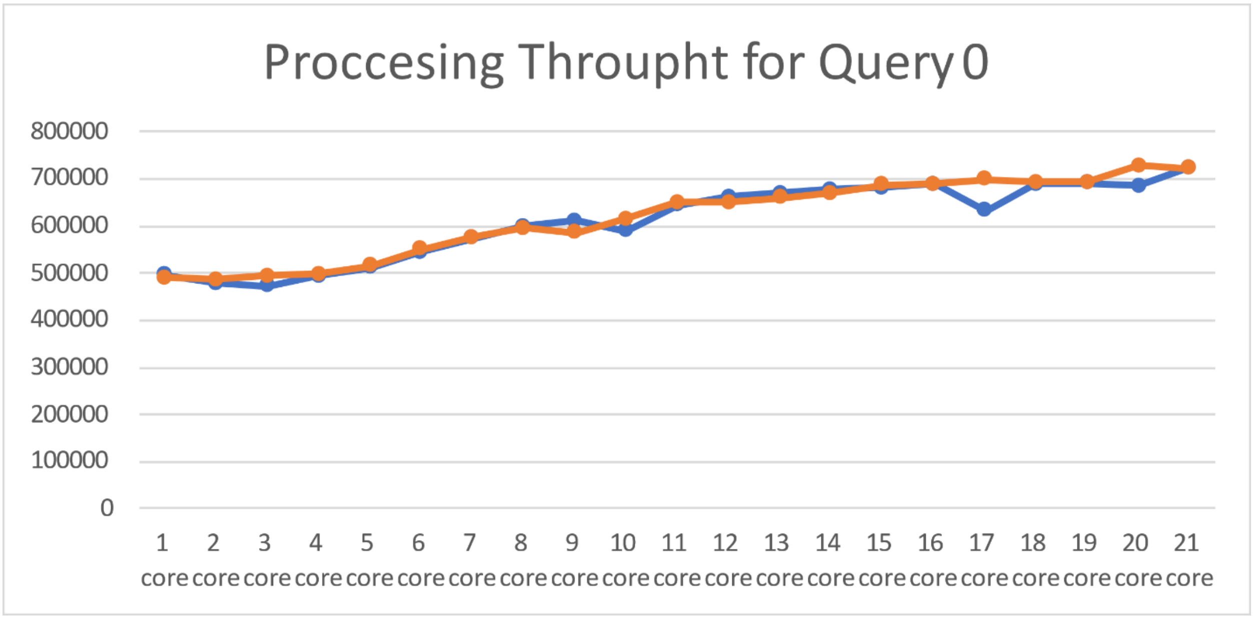


Figure: Throupght for query 0 in batch mode

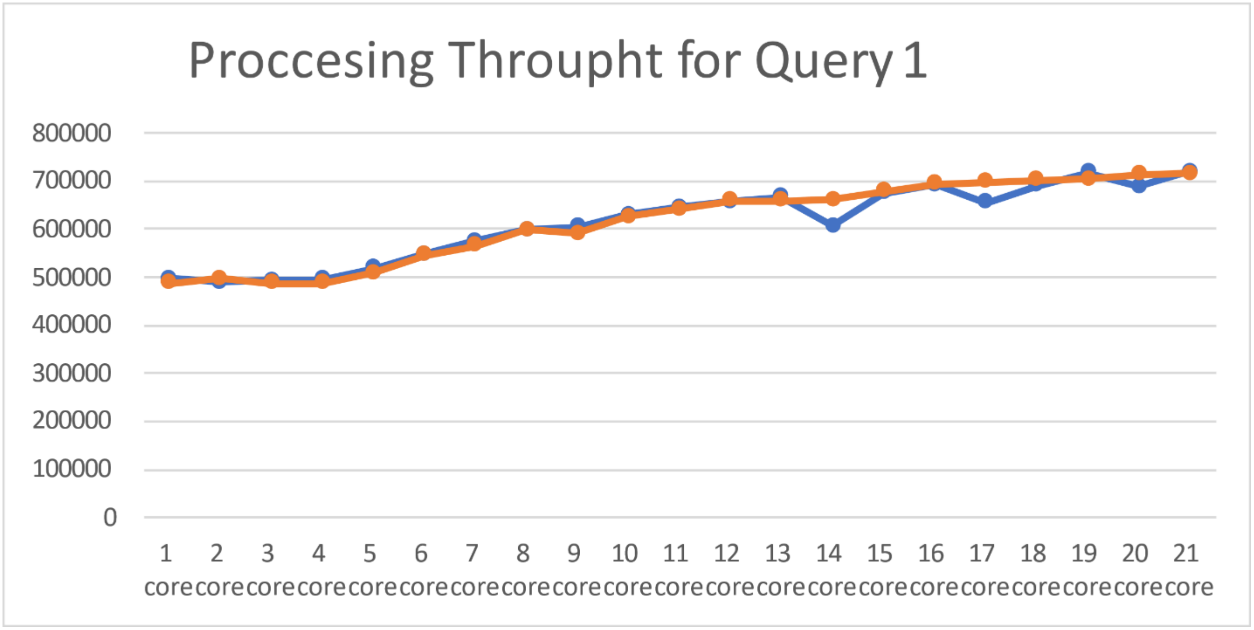


Figure: Throupght for query 1 in batch mode

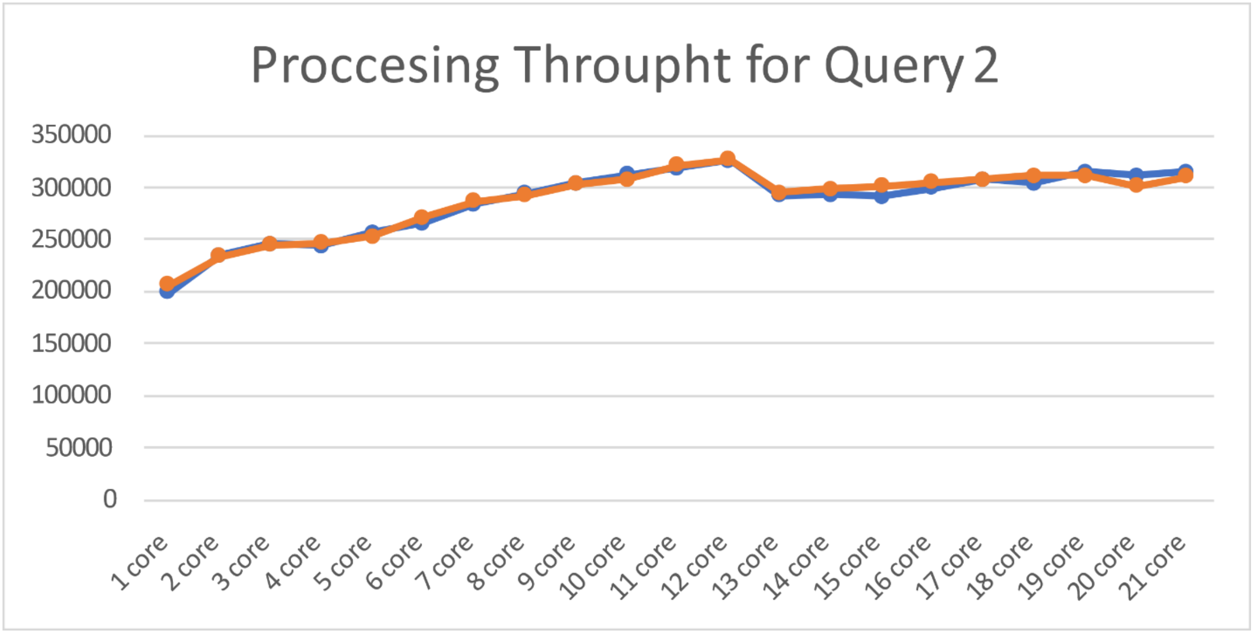


Figure: Throupght for query 2 in batch mode

As we can see here, for each query, we repeat the experiment twice, wherein each time we ran the experiment 21 times every time we increased the number of assigned cores by one.

- And as expected, the throughput was increasing by increasing the number of cores.

- Also, as we can see, the number of RDD partitions increases by increasing the number of cores, which means more tasks in parallel are being handled.
- Nevertheless, the reading time was almost constant every time, which was not expected because as long as the number of cores is increasing, the reading time is supposed to decrease. However, this means we have a sort of bottleneck in our input rate, which, if we solved, we would see significant improvements in our throughput.

streaming mode:

Similar to the batch mode after we finish running the experiments. We draw the throughput that we got for each query, as we can see in the following figures:

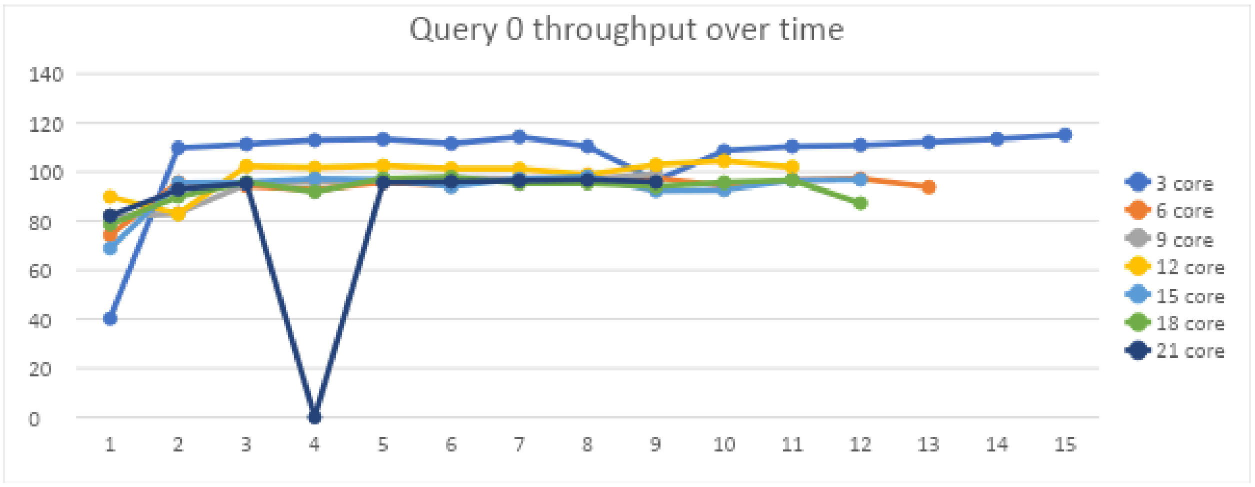


Figure: Throupght for query 0 in stream mode

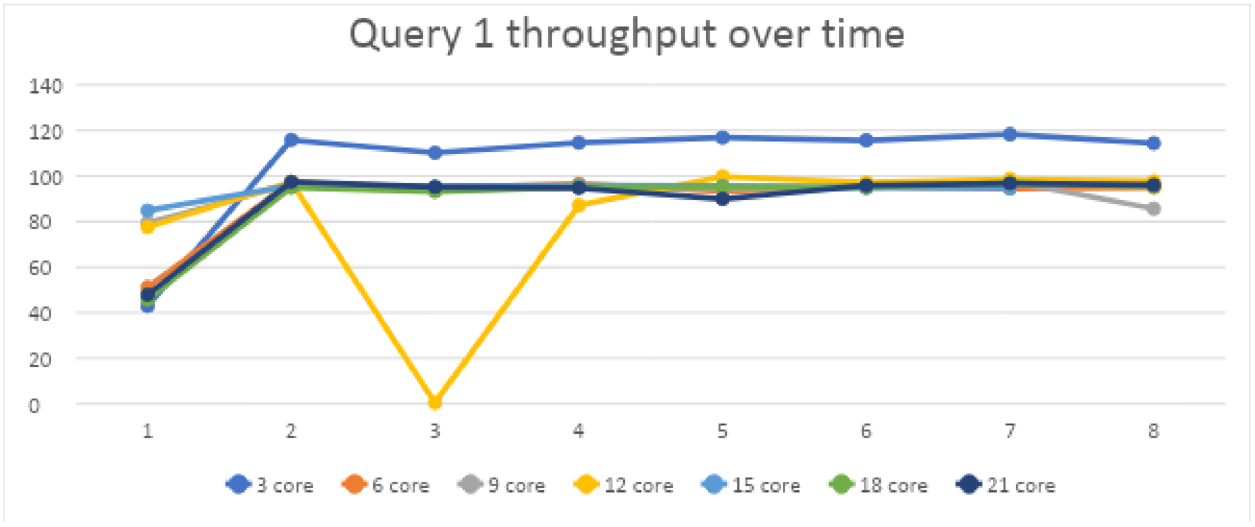


Figure: Throupght for query 1 in stream mode

As we notice from these experiments, the throughput is almost the same regardless of the number of cores. Furthermore, sometimes, there was a significant drop in throughput.

However, this means there is something wrong with the experiment's setup, which we believe is the input rate too low.

Moreover, after reading multiple articles using this method we mentioned before, the result was that the input rate stays the bottleneck. Furthermore, the best solution they said to solve this problem was to generate the events on the fly.

Conclusion

It is almost impossible to get the expected throughput with all parameters correctly tuned in the beginning. It needs some patience and continuation testing to arrive at the correct configurations. It is better to start with incremental tuning. This means writing our code and run it for the first time without changing any of the configurations. We keep the parameters mentioned in the spark doc in our minds. Clearly, things will go wrong in first attempts, like some memory issue, higher processing time, scheduling delay, etc., will come.
At those times, logging correctly in the file will come in handy to debug the root cause, so it is crucial to have good logging. After that, you should Start tuning parameters one by one and keep watching, such as:

- Batch Interval Parameter
- Concurrent Jobs Parameter
- Uniform Data Distribution
- Memory Parameter

Our goal was to understand better the relationship between the number of cores and the throughput of Apache Spark in data processing.

After developing and scaling our applications, we figured out that Apache Spark's distributed applications need meticulous configuration and coding to achieve linear scalability.

The most important thing:

- the reading operation must be done in parallel so that the throughput would increase by increasing the core number
- Cashing the input data help a lot in decreasing spark processing time which is also an excellent factor to increase the throughput.

However, when we ran our experiments:
First, we had some linear scalability in the batch model but not as we expected to get. However, by cashing the input, our scalability improved but still not as we hoped. Also, by changing the memory assigned to each executor, there was no big difference as long as the allocated memory was sufficient to perform the small task assigned.
The final thought was that we still have a reading problem since the reading time was almost constant for all cases which we believe due to the reading was performed locally instead of using the Hadoop cluster to read from it which we are planning to test in our future research.

Second, in the streaming model, as we processed using micro-batching, there was no linear scalability at all. Sometimes, there was a considerable drop in the input rate, which also influenced the throughput rate. We apply the same enhancements we did in batch mode, but still, the results are the same, which tells the input rate is the bottleneck for the system.

Trying the different techniques to parallelism the reading, hoping to increase the input rate, unfortunately, there were no excellent results. We believe the only option left for us is to generate the events on-fly similar to the Apache Beam project, which we plan to do in future research.

We believe our research could be so helpful as a starting point for any researcher interested in doing more complex analysis using Apache Spark and wants to add more metrics to compare.

References

1. https://spark.apache.org/docs/latest/
2. https://beam.apache.org/documentation/sdks/java/testing/nexmark/
3. Tucker, P.A., Tufte, K., Papadimos, V., & Maier, D. (2002). NEXMark – A Benchmark for Queries over Data Streams DRAFT.
4. J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," 2018 IEEE 34th International Conference on Data Engineering (ICDE), 2018, pp. 1507-1518, DOI: 10.1109/ICDE.2018.00169.
5. Stream Processing with Apache Spark: Best Practices for Scaling and Optimizing Apache Spark by Gerard Maas, Francois Garillot
6. Mastering Apache Spark: Gain expertise in processing and storing data by using advanced techniques with Apache Spark by Mike Frampton
7. O. Marcu, A. Costan, G. Antoniu and M. S. Pérez-Hernández, "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks," 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp. 433-442, doi: 10.1109/CLUSTER.2016.22.
8. García-Gil, D., Ramírez-Gallego, S., García, S. *et al.* A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Anal* 2, 1 (2017).
9. M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, “Scaling spark in the real world: Performance and usability,” Proc. VLDB Endow., vol. 8, no. 12, pp. 1840–1843, Aug. 2015.
10. R. Tous et al., "Spark deployment and performance evaluation on the MareNostrum supercomputer," 2015 IEEE International Conference on Big Data (Big Data), 2015, pp. 299-306, doi: 10.1109/BigData.2015.7363768.
11. R. Souza, V. Silva, P. Miranda, A. Lima, P. Valduriez, and M. Mattoso, "Spark Scalability Analysis in a Scientific Workflow," in SBBD 2017: 32th Brazilian Symposium on Databases, Uberlandia, Brazil, Oct. 2017,