DATA+AI SUMMIT 2022

Improving Spark Structured Streaming Application Processing Time

By Configurations, Code Optimizations and Custom Data Source

Kineret Raviv
Principal Software Developer, Akamai

Nir Dror
Principal Performance Engineer, Akamai

ORGANIZED BY databricks

About Us

Kineret Raviv

- Principal Software Engineer at Akamai
- 8+ years of experience in Big data technologies (Spark, Hadoop, Map-Reduce)

Kineret Raviv

Nir Dror

- Principal Performance Engineer at Akamai
- Tuning and troubleshooting performance issues in Spark applications since 2016

Nir Dror

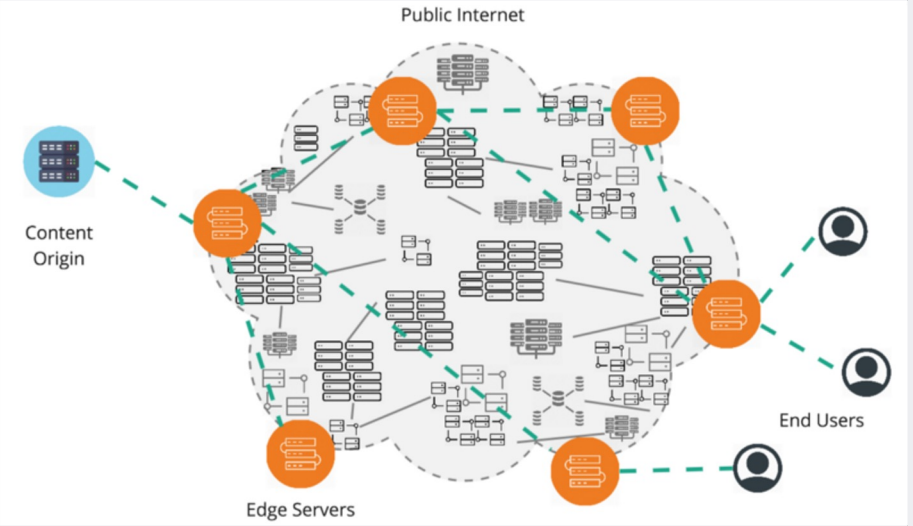
Agenda

- Overview
- Custom Spark Data Source
- Performance Tuning
- GC Analysis

Overview

About Akamai Technologies

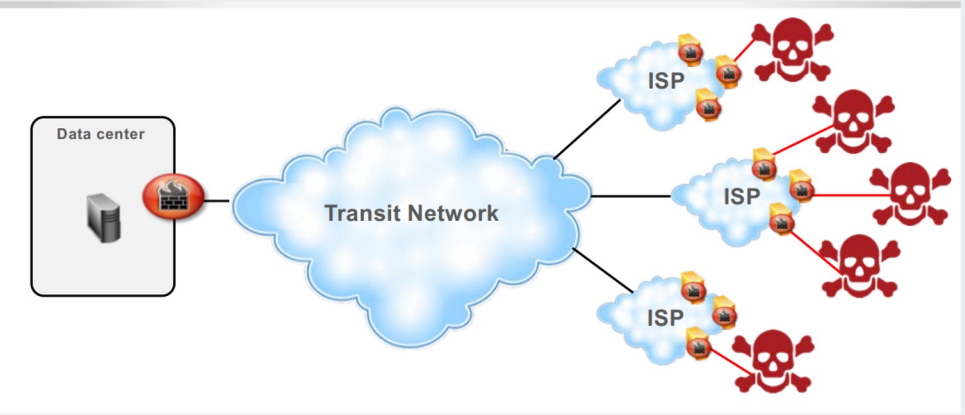
- Largest CDN services provider and cloud security solutions
- 350K servers
- **8B requests per day**
- **~ 30% of the global internet traffic**



Challenge: Dealing with a massive amount of data

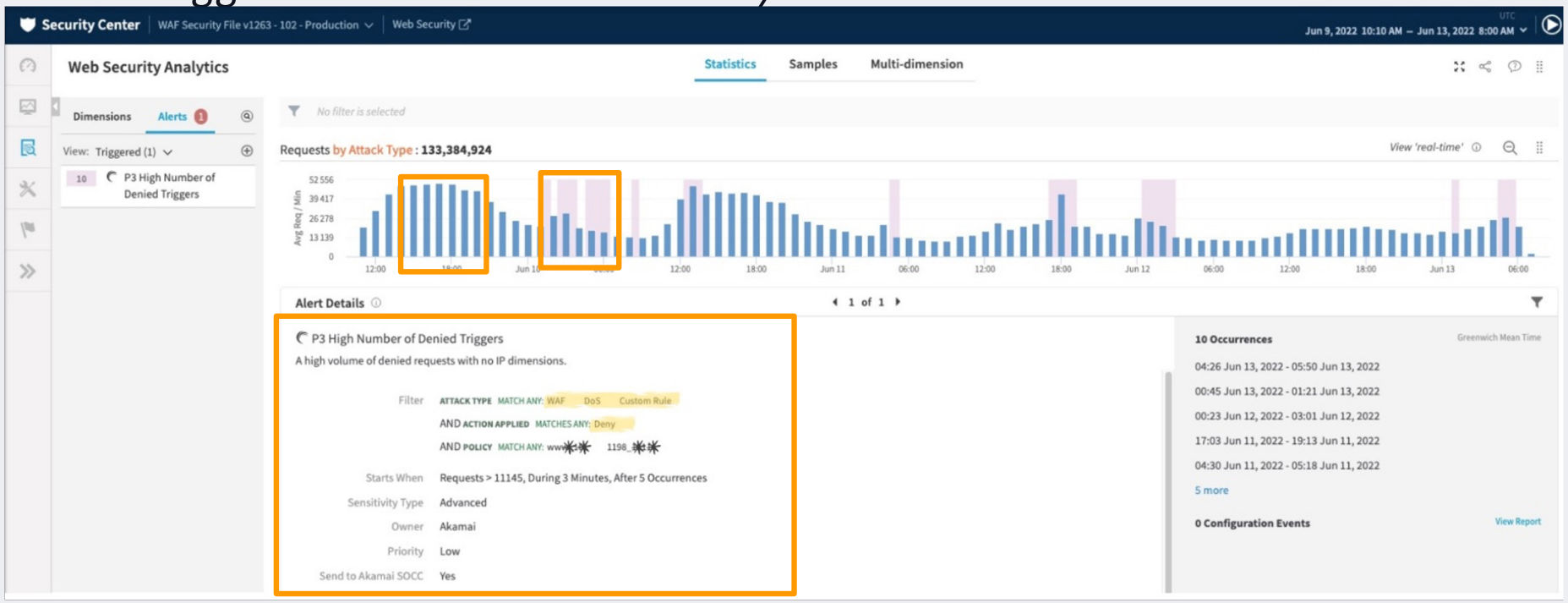
About CSI Group (Cloud Security Intelligence)

- Platform for collecting, analyzing, and distilling quality security intelligence information
- **~ 6GB/s incoming traffic**
- **~ 100B events per day**

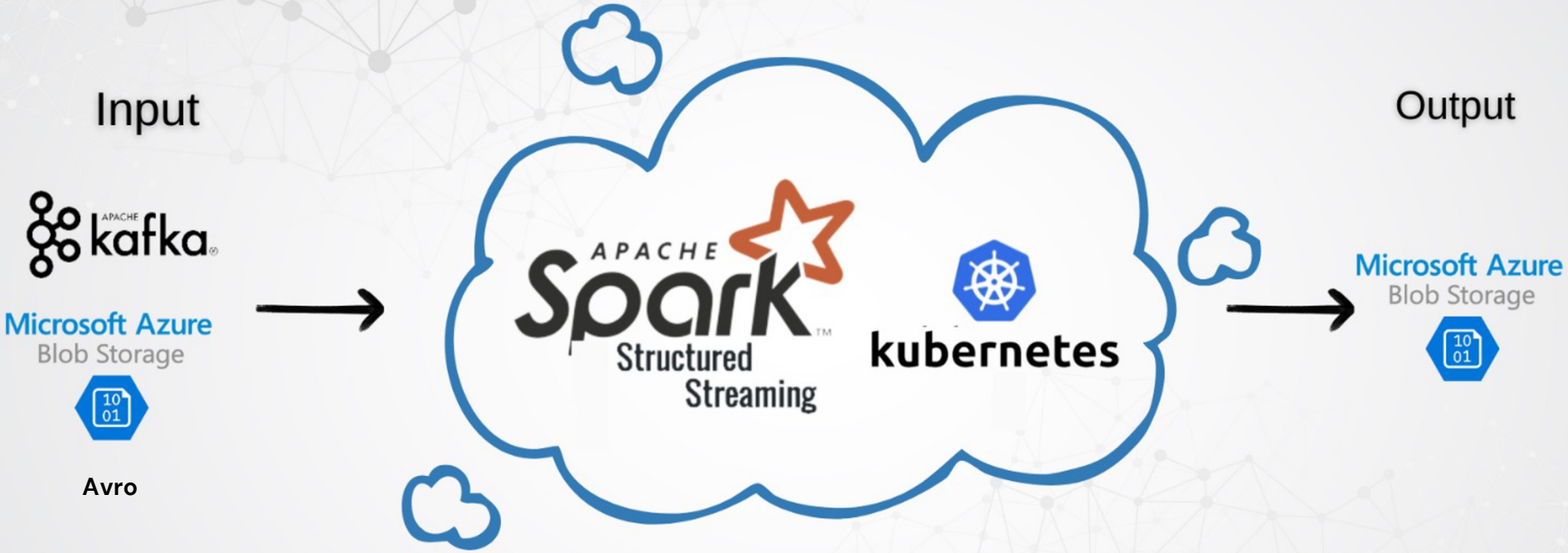


The product

Goal: *Trigger alerts based on security events customer's definitions*



Application Architecture and Overview



Spark resources: **Driver** - 1 core, 4G memory,
Executors: 200 instances, 2 cores, 2G memory

Other utilities: **Java 11, Spark-on-k8s-operator**

Spark Data Source

Input Architecture

kafka

Queue with “pointers” to the blob and metadata. For example:

```
{
  "path": "/2022-06-13/1655112426123-waf-triggers-
dlr/9a38250e-5210-476e-9fcd-e05815df7896.avro.deflate",
  "size": 7526435,
  "recordsCount": 9686,
  "statistics": {"1655110800000": 9686}
},
{
  "path": "/2022-06-13/1655112419993-waf-triggers-
dlr/95047b9f-14cb-48a3-a015-98784534faaa.avro.deflate",
  "size": 10586605,
  "recordsCount": 14750,
  "statistics": {"1655110800000": 14750}
}
```

Microsoft Azure
Blob Storage

The actual avro files

Read Phase: Spark Data Source Overview

The **Data Source API** allows us to read data from different sources in a distributed way.

- Built in standard sources (json, parquet, jdbc, orc, libsvm, csv, text):

```
spark.read.format("parquet").load("myFile.parquet")
```

```
spark.read.json("myFile.json")
```

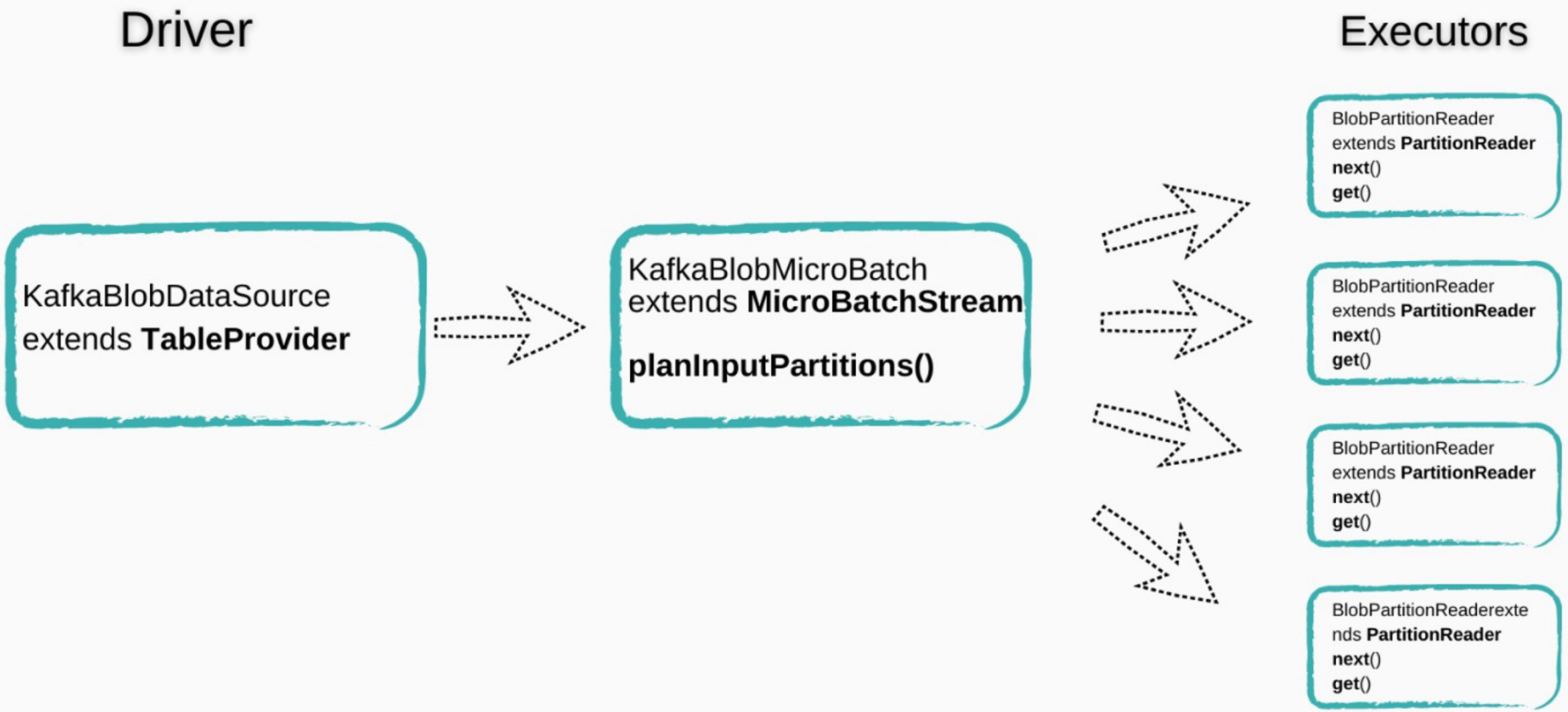
- Third party sources by extending this API

```
spark.read.format("org.apache.spark.sql.cassandra")
```

- Our custom data source

```
spark.readStream().format("com.akamai.csi.connectors.KafkaBlobDataSource")
```

Spark Data Source Implementation



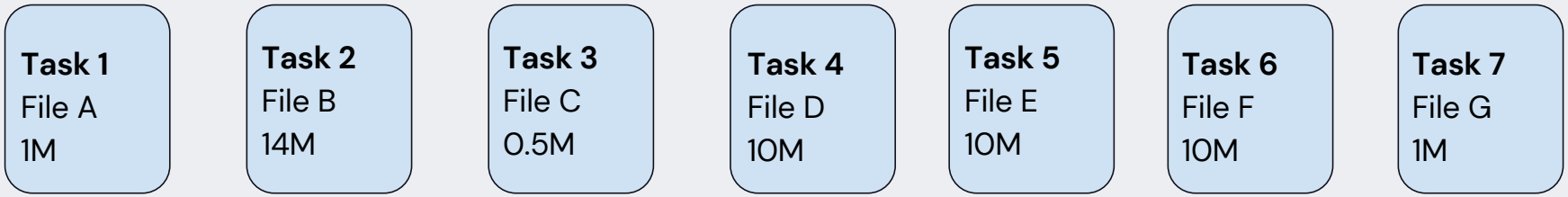
PlanInputPartitions()

- Initialize Kafka consumer (in class level)
- In each iteration poll messages until:
  - *Catch Up mode (high kafka lag)* - the total microbatch memory provided by the user is reached (sum on “size” property)
  - *Regular mode* - the queue is empty
- **Optimization 1: filter message according to TTL – saves I/O**
- Divide the messages to partitions
- **Optimization 2: Partitioning Strategies**

Partitioning Strategies

Goals: Efficiency and avoiding data skew

- Single file per partition

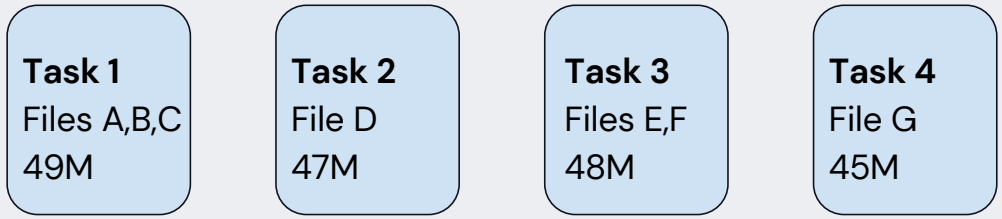


High number of tasks

Partitioning Strategies

Goals: Efficiency and avoiding data skew

- Multiple files by size of partition

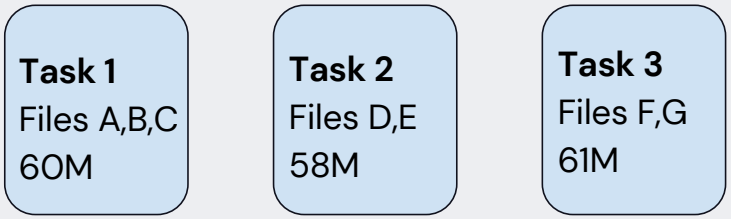


# of tasks is not fixed => Resources are not fully used

Partitioning Strategies

Goals: Efficiency and avoiding data skew

- Multiple files by number of tasks



Optimal ✓

Dynamic number of tasks

Challenge: Reduce throttling errors

Solution: read parallelism according to Kafka lag

Example: 400 available cores

```
val numOfPartitions = if (lag > 20K)  2000 else 300
```

Custom Spark Data Source – Summary

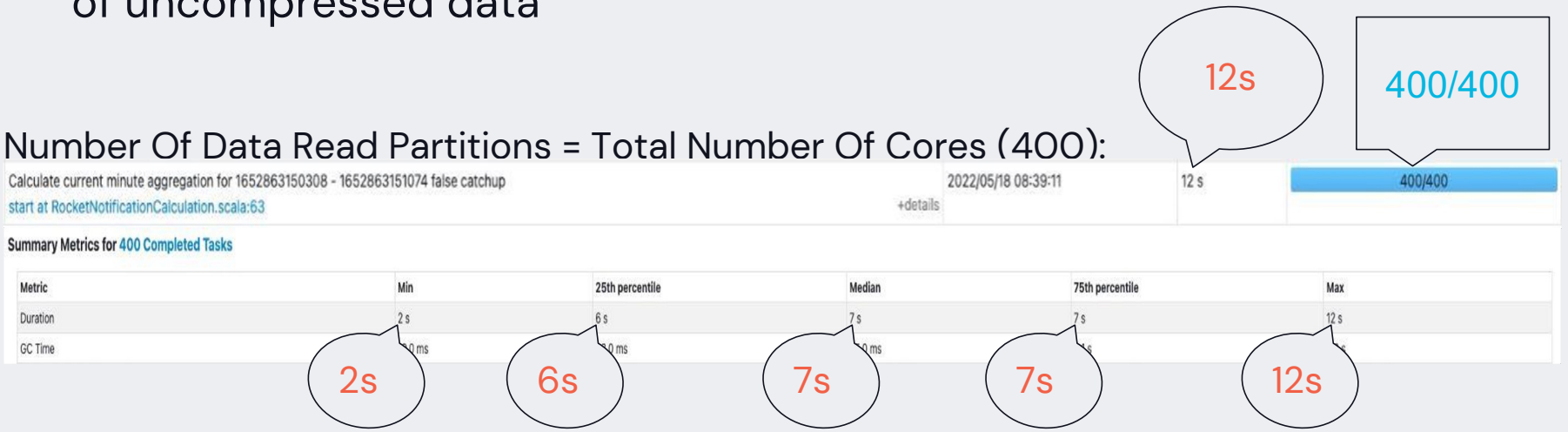
- Try to read only the relevant data in the executors
- Plan partitions equal in size as much as possible
- If the source supports, implement supportsPushDownAggregates/ SupportsPushDownFilters/SupportsPushDownRequiredColumns interfaces

Performance Tuning & GC Analysis

Optimal Number of Partitions

- Challenge: Choosing the optimal number of partitions for reading 100 GB of uncompressed data

Number Of Data Read Partitions = Total Number Of Cores (400):

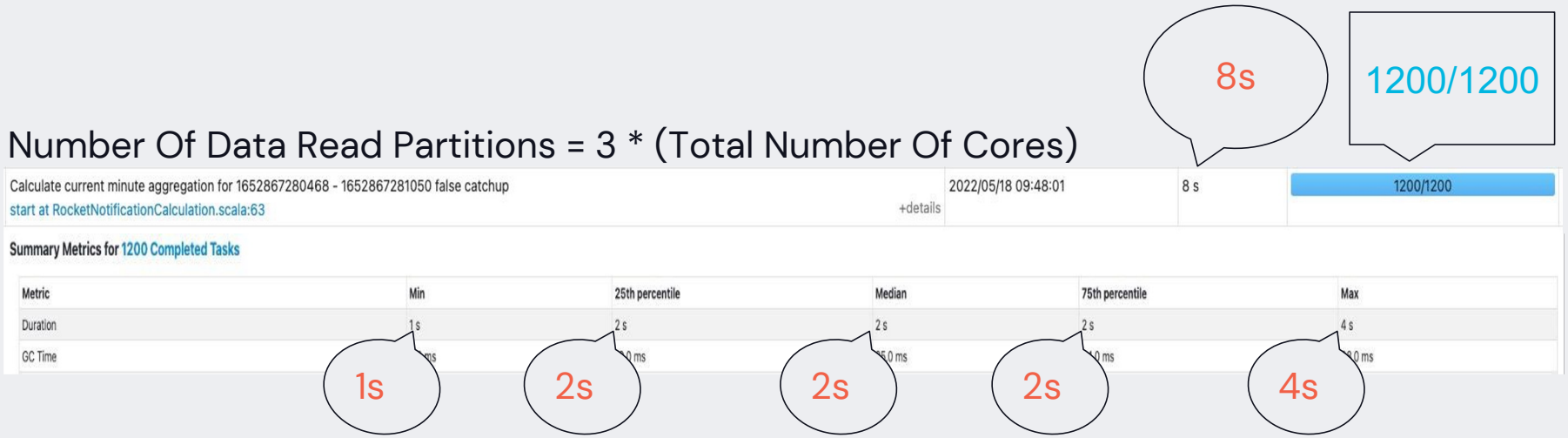


Optimal Number of Partitions

Number Of Data Read Partitions = 2 * (Total Number Of Cores)

Optimal Number of Partitions

Number Of Data Read Partitions = 3 * (Total Number Of Cores)



Optimal Number of Partitions

- Conclusion: Creating smaller (balanced) tasks by increasing the number of partitions we used to read the data (to be higher than the number of cores we have) helped to reduce the read time by almost 50%!

Garbage Collection

- Enabling GC logs (and some more useful information):
  - Java Version <= 8:

    ```
    -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -
    XX:+PrintTenuringDistribution
    ```

  - Java Version > 8:

    ```
    -Xlog:gc*,gc+ref=debug,gc+heap=debug,gc+age*=trace:file=<gc-file-path>
    :tags,uptime,time,level
    ```

Garbage Collection – Analysis

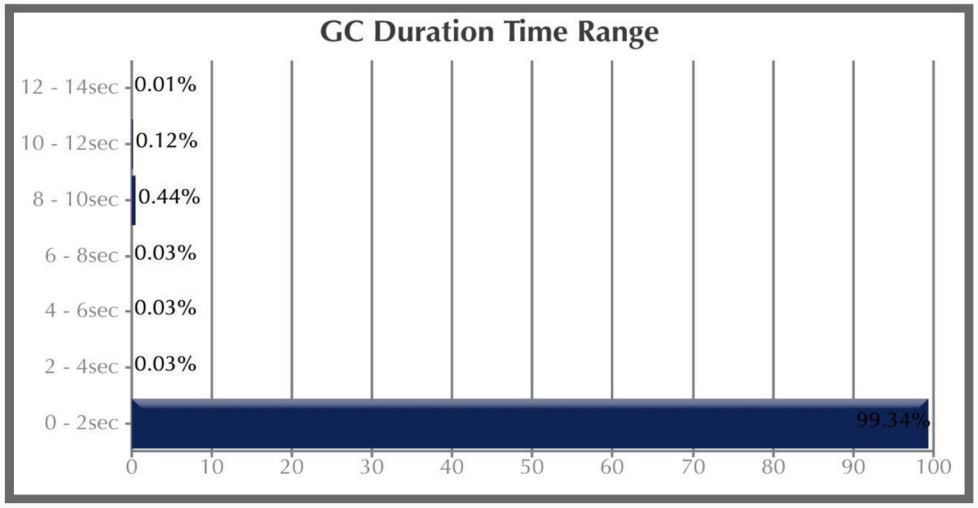
1 Throughput : 95.811%

2 Latency:

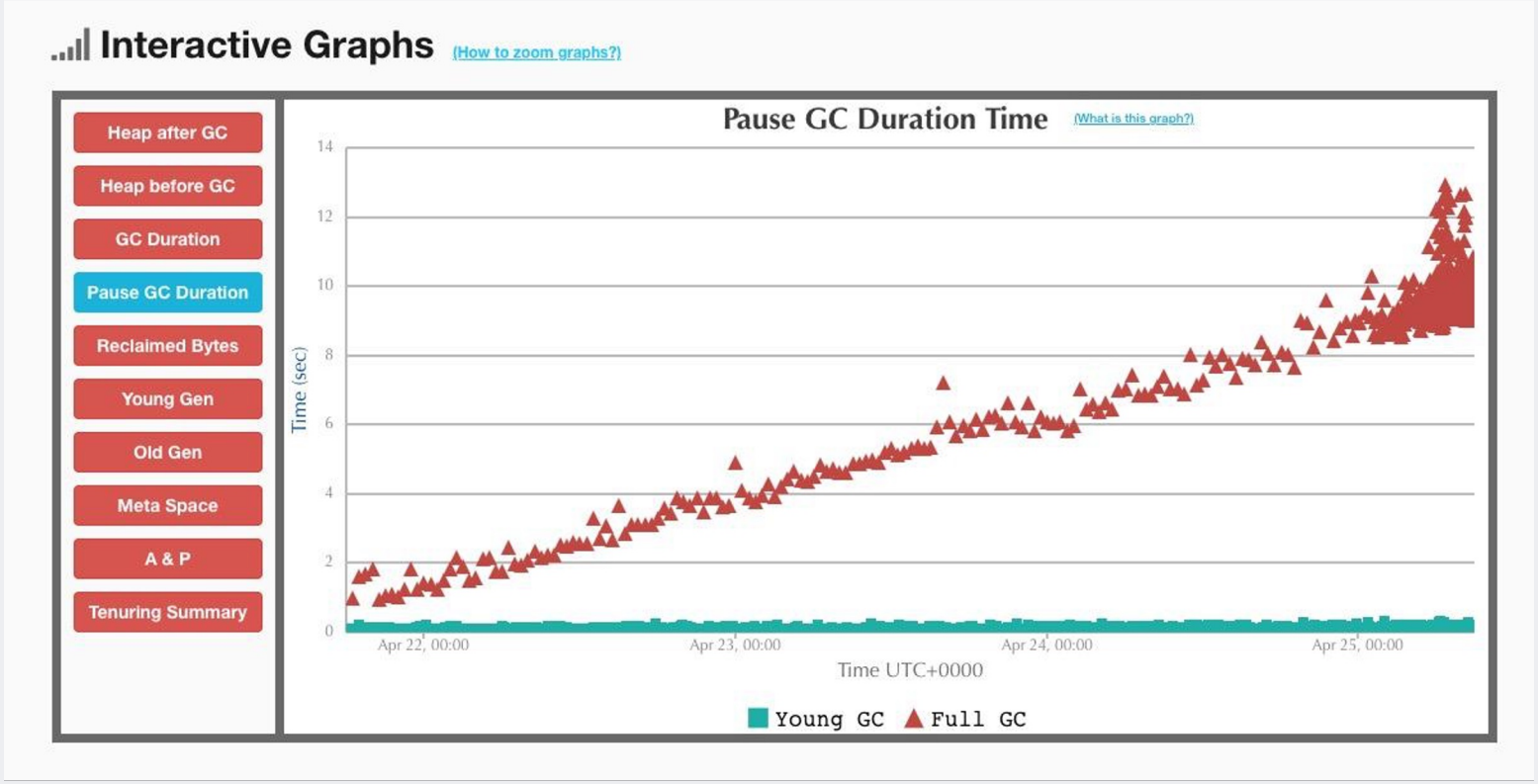
Avg Pause GC Time	97.2 ms
Max Pause GC Time	12 sec 930 ms

GCPauseDuration Time Range :

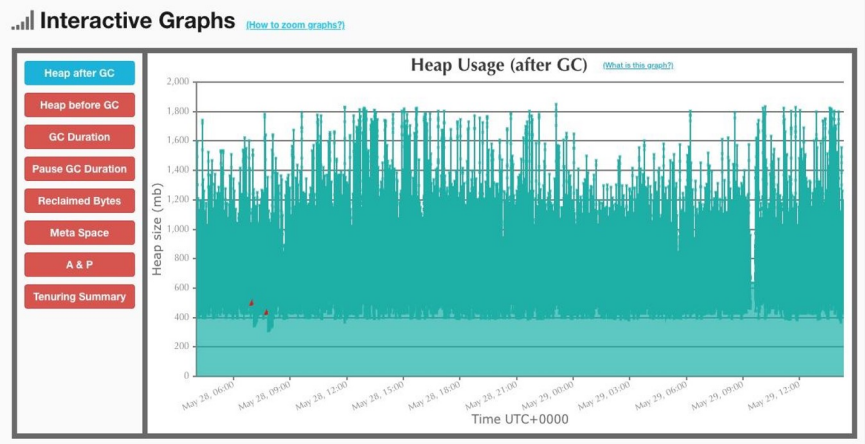
Duration (sec) 2 sec Change	No. of GCs	Percentage
0 - 2	133943	99.34%
2 - 4	40	0.03%
4 - 6	36	0.03%
6 - 8	41	0.03%
8 - 10	596	0.44%
10 - 12	165	0.12%
12 - 14	11	0.01%

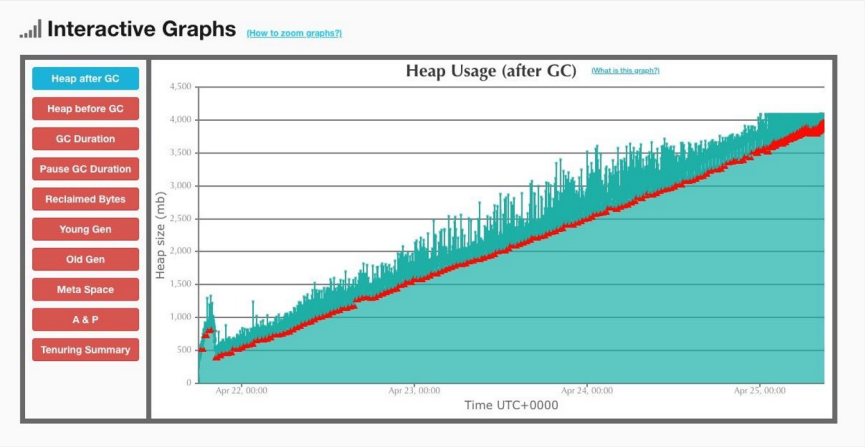


Garbage Collection – Analysis



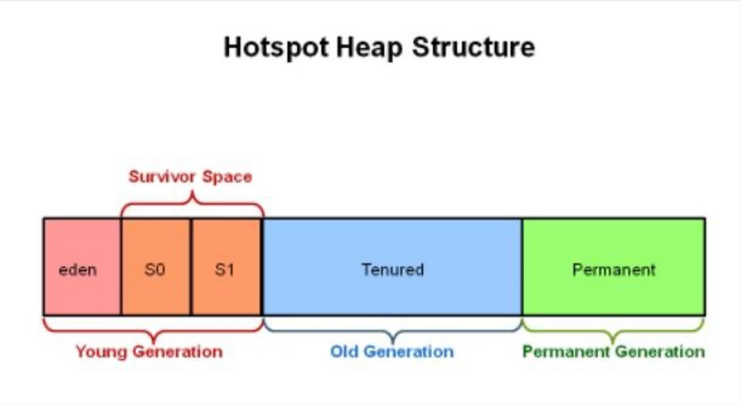
Garbage Collection – Analysis





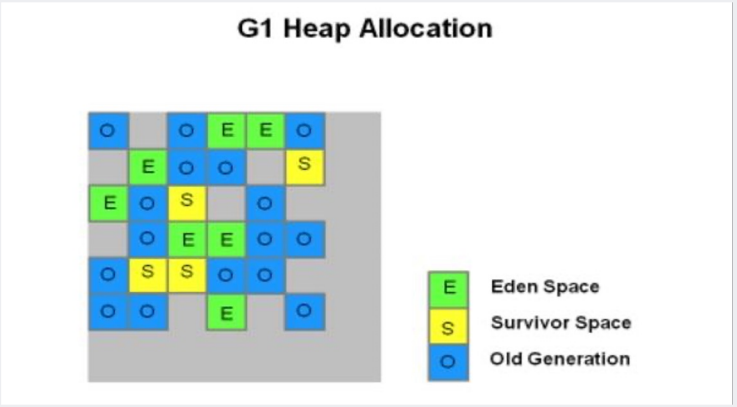
Garbage First (G1) GC

- Default GC from Java >= 9
  To enable it in previous versions, add the following to your driver / executors: `-XX:+UseG1GC`



Heap Structure Before G1

(Photos by oracle.com)



Garbage First (G1) GC

- Required much more tuning when using it in earlier Java versions. For example:

```
-XX:+UseG1GC
-XX:ConcGCThreads=5
-Xms12g
-XX:NewSize=6g
-XX:MaxTenuringThreshold=5
```

- Works faster in Java >= 10 because of the ability to use multiple threads for full GC

- Using G1 GC in newer Java versions can boost your application's performance

Garbage Collection - Summary

- Adding GC logs to your Spark application is very simple!
- Analyzing your application's GC logs can help to identify memory issues
- Important metrics to notice when you analyze GC logs:
  - Throughput
  - Average & Max GC pause time
  - Heap size after GC
- G1 GC is more flexible in terms of memory usage compared to the older Garbage Collectors. Consider using it (especially in Java >=10) if possible

DATA+AI SUMMIT 2022

Thank You

Kineret Raviv
Principal Software Developer, Akamai

Nir Dror
Principal Performance Engineer, Akamai