ANALYSIS OF COST/RESOURCE OPTIMISATION IN AWS EMR

After going through the Official Amazon AWS Management Guide for EC2 servers, I found certain optimisation techniques- in terms of cost and/or process that I have specified below.

I am unaware if these are already in implementation but i have presented them here in case they are not and in your opinion you might find them to be useful:

- *Storage of Different File Systems within the Cluster:*
  Use HDFS instead of EMRFS- it is more useful for caching intermediate results during parallelizable processes across large datasets using MapReduce, similar to the case I believe we need for processing data during transformation of data from the backup DB.
- *Spot Instances:*
  In terms of cost optimisation, it can be used for less frequently run instances because it uses the spare EC2 capacity available at a lesser price. There is an elevated risk of interruption but a process is interrupted only if the memory space is no longer available.
- I have no idea if CDNs are being used but I believe that can be used to optimize the servers.

Potential problems that might be occurring and causing the processing time to have increased:

- Cluster configurations that were set at the creation can only be overridden, not deleted. Thus, if any differences are found between the existing configuration and the file that is supplied, amazon EMR resets the manually modified configurations to the cluster defaults for the specified instance group.
- No change in yarn-site and capacity scheduler properties, which if manually changed can help in taking advantage of the node labels.
- On demand container allocation instead of Standard/Convertible Reserved Containers

Type of Instances being Used:
Instance being used for master and core as of right now is r5 which is a memory optimized instance type. Since data is accessed after a day from the backup DB, storage optimized instances should be explored.

Features of Storage Optimized Instances-

- Facilitate rapid, sequential read and write access to vastly large datasets present on local storage
- Allow applications to perform multiple random IOPS at low latency

STORAGE OPTIMIZED COMPARISON

Data is accessed after a day from the backup DB anyway so why not go for storage optimized instances

Param	I4g	Im4gn	Is4gen	D2
Price	Best in this domain	40% better	48% better	Lowest price per disk throughput performance
Processor	AWS Grav2	AWS Graviton 2	AWS grav 2	High freq Intel Xeon Scalable Processors
Memory	15TB of NVMe SSD with AWS Nitro SSD	44% lower cost per TB, 30 TB NVMe SSD+ AWS Nitro SSDs	15% lower cost per TB, lowest per TB and highest SSD density per vCPU, 30 TB+ AWS	48 TB HDD based local storage
Features	Optimised for workloads that map 8 GB of memory per vCPU	Optimised for workloads that map 4 GB of memory per vCPU	Optimised for workloads that map 6 GB of memory per vCPU	Intel AVX and Intel Turbo
	Support for TWP to facilitate addn performance and reduce latencies and DB workloads such as MySQL	Support for TWP to facilitate addn performance and reduce latencies and DB workloads such as MySQL	Support for TWP to facilitate addn performance and reduce latencies and DB workloads such as MySQL	Consistent high performance at launch time, high disk throughput
	EBS Optimized	EBS Optimized	EBS Optimized	EBS optimized
	Enhanced Networking	Enhanced Networking	Enhanced Networking	Enhanced Networking
Uses	I/O intensive applications, targeted to customers using transactional databases, real time analytics such as Apache Spark	Maximizes number of TPS for I/O intensive for medium size datasets that benefit from high compute performance and RDBMS // data analytics workload	Maximizes number of TPS for I/O intensive for large size datasets, workloads with higher storage density, very fast access to datasets(large distributed file systems)	MPP data warehousing, Map Reduce and distributed computing system between files and network or log processing applications

Instance Type	Instance Size	vCPUs	Memory	On-Demand Hourly Cost	Cost for 4 instances
r5	8xlarge	32	256 GiB	2.137 USD	6240.04 USD
i4g	8xlarge	32	256 GiB	2.718 USD	7936.96 USD

While the pricing of these instances is quite high, given the additional features for I/O intensive applications and their database computations, they can be taken into consideration by you.

Following assumptions were made based on information made available by the team to me for pricing calculation:

1. 1 instance for Master and 3 instances for Core are being used.
2. Cost has been calculated using the AWS EMR pricing calculator with the following specifications: Tenancy-Dedicated Instances Operating System- Linux Workloads- Constant Usage
3. On Demand Containers are being used.
4. Cost has been calculated for 730 hours and the Cost for 4 instances column DOES NOT contain the Dedicated Per Region Fee= 730 hours x 2 USD

Attached is a feature comparison of all memory optimized instances that have been rolled out since r5 and I found relevant to the requirements of the company at this time.

MEMORY OPTIMIZED COMPARISON

Param	R8g	R7g	R7i	R6g	R6a	R5
Price Performance	*	High	15% better than R6i	40% better than R5	35% better than R5a	
Processor	AWS Graviton4	AWS Graviton3	4th Gen Intel Xeon Sc 3.2 GHz	Arm based AWS Grav2	3rd gen AMD EPYC 7R13	3.1 GHz Intel Xeon Plat 8000 with AVX 512
Memory	DDR5-5600	DDR5	DDR5			768 GiB of memory per instance
Features	EBS optimized	EBS optimized	Support for up to 128 EBS vol attch/inst // EBS opt	EBS optimized		EBS optimized
		Enhanced networking	Enhanced networking	Enhanced Networking		Enhanced Networking
Extra special feature	Larger instances		Has AMX, always on memory encryp using TME, built in accelerators* DSA IAA QAT		Support on always on memory encryp AMD TSME And AMD AVX2 for faster execution of algo	
	AWS Nitro System	AWS Nitro Sys	AWS Nitro System	AWS Nitro System	AWS Nitro System	AWS Nitro System
Uses	Open source DBs, real time big data analytics	Open source DBs, real time big data analytics	Memory intensive, distributed web scale, real time big data analytics	Open source DBs, real time big data analytics	Enterprise applications in general	Distr web scale in memory caches, mid size in memory DBs

Have not included the instances that have advantages and applicability for real time databases

Param	X1e	X2iezn	X2iedn	X2gd
Price	One of the lowest prices per GiB of RAM	55% better than X1e	50% better than X1	55% better than X1, the lowest cost per GiB
Memory	3904 GiB of DRAM based instance memory	32:1 ratio of memory	32:1 Ratio of memory to vCPU	
Processor	High freq Intel Xeon E7-8880 v3	4.5 GHz 2nd gen Intel Xeon Scalable	3.5 GHz 3rd gen Intel Xeon Sc	Arm based AWS Graviton2, 64 bit Neoverse cores
Features	SSD instance storage for temporary block level storage	Fastest processor	Support for always on mem encryp TME+ AVX 512	
	EBS optimized	EBS optimized	EBS optimized	EBS optimized
	Enhanced Networking	Enhanced Networking	Enhanced Networking	Enhanced Networking
	Intel AVX and *Turbo	AWS Nitro System	AWS Nitro System	AWS Nitro System
Uses	High performance databases, in memory databases	Electronic design automation like physical verification, static time an	Large scale in memory/trad databases, in memory analytics	Real time caching servers, memory intensive workloads

As clearly indicated among the r-line, r7i has multiple mathematical abilities in-built that will help in faster processing of the data and potentially lead to lesser computation time overall. In terms of pricing:

Instance Type	Instance Size	vCPUs	Memory	On-Demand Hourly Cost	Cost for 4 instances
r5	8xlarge	32	256 GiB	2.137 USD	6240.04 USD
r7i	8xlarge	32	256 GiB	2.328 USD	6799.99 USD
r6a	8xlarge	32	256 GiB	1.995 USD	5827.85 USD
r6g	8xlarge	32	256 GiB	1.709 USD	4992.03 USD

As confirmed by multiple technical blogs and more, the only way to achieve cloud spending efficiency is to use fixed memory sizes for your executors that achieve **optimal CPU utilization**.
[Link to the referred technical blog]

EXCEL SPREADSHEET:

EMR configuration

Note: Inputs inserted are sample values for reference
1) Total Ram instance used is the yarn.nodemanager.resource.memory-mb based on task configuration of instance type
2) spark.executor.cores, number of core instances are inputs based on which rest calculation will be done. Based on instance type, we need to provide virtual cores available and total RAM

Calculations based on best practices
1. Number of executors per instance = (total number of virtual cores per instance - 1)/ spark.executors.cores
(subtracting one virtual core to reserve for Hadoop daemons)
2. Total executor memory = total RAM per instance / number of executors per instance (rounded down)
3. spark.executors.memory = total executor memory * 0.90 (10% assigned for memory overhead) (rounded down)
4. spark.yarn.executor.memoryOverhead = total executor memory * 0.10(rounded up)
5. AWS recommendation: spark.driver.memory = spark.executors.memory
6. AWS recommendation: spark.driver.cores= spark.executors.cores.
7. spark.executor.instances = (number of executors per instance * number of core instances) minus 1 for the driver

On reviewing the documentation I didn’t find any other formulae that would better suit the fixing of these parameters.

As suggested before, fine-tuning of YARN resource manager configurations based on workload requirements for better resource utilization will help optimize the computation even more.

Enable speculative execution to mitigate slow tasks: spark.speculation=true
Also ensure
spark.dynamicAllocation.enabled=true and fix the minimum and maximum number of executors.