Εθνικο Μετσοβιο Πολυτεχνειο
ΔΠΜΣ Επιστημη Δεδομενων και Μηχανικη Μαθηση

Αποτίμηση της Κατανεμημένης Διαχείρισης Κατάστασης σε Συστήματα Διαχείρισης Δεδομένων Μεγάλης Κλίμακας

Διπλωματικη Εργασια

Κωνσταντινος Πλευρης

Αθήνα, Απρίλιος 2022

National Technical University of Athens
Data Science and Machine Learning

Evaluation of Distributed State Management in Big Data Systems

Master Thesis

Konstantinos Plevris

Athens, April 2022

Εθνικο Μετσοβιο Πολυτεχνειο

ΔΠΜΣ Επιστημη Δεδομενων και Μηχανικη Μαθηση

Αποτίμηση της Κατανεμημένης Διαχείρισης Κατάστασης σε Συστήματα Διαχείρισης Δεδομένων Μεγάλης Κλίμακας

Διπλωματικη Εργασια

Κωνσταντινος Πλευρης

Επιβλέποντες: Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.
Ιωάννης Κωνσταντίνου Επίκ. Καθηγητής Παν. Θεσσαλίας

Εκρίθηκε από την τριμελή εξεταστική επιτροπή την 11^{η} Απριλίου 2022.

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
Ιωάννης Κωνσταντίνου	Νεκτάριος Κοζύρης	Γεώργιος Γκούμας
Επίκ. Καθηγητής Παν. Θεσσαλίας	Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2022

Περίληψη

Αυτή η διπλωματική εργασία έχει ως αντικείμενο την απόπλεξη των ενδιάμεσων δεδομένων που προκύπτουν από τον μηχανισμό του shuffle, καθώς είναι γνωστό πως αυτό αποτελεί σημείο συμφόρησης κατά την επεξεργασία τους. Θα εστιάσουμε ειδικότερα στο εργαλείο Apache Spark, το οποίο χρησιμοποιείται για την επεξεργασία δεδομένων μεγάλης κλίμακας.

Παρουσιάζονται δύο τρόποι αποθήκευσης των ενδιάμεσων αποτελεσμάτων του Shuffle 1) Με το redis (το οποίο αποτελεί ένα cache που αποθηκεύει δεδομένα στην μνήμη (RAM)) 2) Με την βάση δεδομένων mongodb που παρατάσσεται σε κατανεμημένη αρχιτεκτονική (Mongodb Distributed Sharded Cluster). Έτσι οι worker (οι κόμβοι του Apache Spark που εκτελούν την επεξεργασία των δεδομένων) αποπλέκονται από την αποθήκευση και διαχείριση των shuffle δεδομένων, καθιστώντας την ύπαρξη τους εφήμερη, καθώς η διατήρηση των δεδομένων δεν γίνεται πλέον σε τοπικό επίπεδο.

Ως κύριο μέσο για τον έλεγχο, την ανάπτυξη και την εξαγωγή αποτελεσμάτων χρησιμοποιήθηκε η γνωστή πλατφόρμα Kubernetes, πάνω από ένα cluster εικονικών μηχανών (virtual Machines VMs). Με αυτό πετυχαίνουμε την εύκολη διαχείριση των κόμβων του Spark, του Redis και του Mongodb. Τέλος, εκτελούμε πειράματα χρησιμοποιώντας ένα τεστ κόπωσης (stress test) για τον μηχανισμό του shuffle έτσι ώστε να αξιολογήσουμε τις υλοποιήσεις μας, όπως και του Vanilla Apache Spark.

Λέξεις κλειδία: Μεγάλα Δεδομένα, Απόπλεξη Δεδομένων, Μηχανισμός Διαμοιρασμού και Ταξινόμησης, Ὑπολογιστικά Νέφη, Κατανεμημένες Βάσεις Δεδομένων, Apache Spark, Redis, Mongodb, Kubernetes

Abstract

The purpose of this thesis is to disaggreagete the intermediate data resulting from the shuffle mechanism as it is known to be a point of congestion during data processing. We will focus in particular on the Apache Spark tool, which is used for large-scale data processing.

We present two ways to store Shuffle intermediate results 1) with redis (which is a cache that stores data in memory(RAM)) 2) with the mongodb database deployed as a distributed architecture (Mongodb Distributed Sharded Cluster). Thus the workers (the Apache Spark nodes that perform the data processing) are untangled from the storage and management of the shuffle data, making their existence ephemeral, as the data is no longer stored locally.

The well-known Kubernetes platform was used as the main tool for controlling, deployment and exporting results, on top of a set of virtual Machines VMs , with which we achieve easy management of Spark, Redis and Mongodb nodes. Lastly, we perform experiments using a stress-testing workload for the shuffle mechanism, in order to evaluate the performance of the redis and mongodb implementations as well as the performance of vanilla Spark.

Keywords: Big Data, Data Disaggregation, Shuffling Mechanism, Cloud Computing, Distributed Databases, Redis, Mongodb, Kubernetes, Shuffling Mechanism

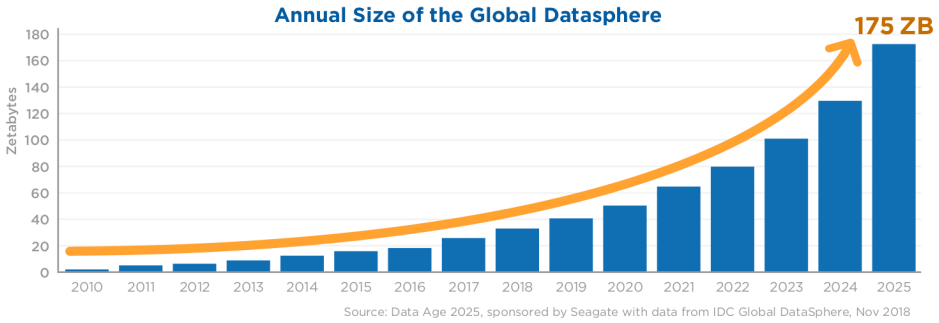
Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Στην σημερινή εποχή, ηλεκτρονικές συσκευές όπως το κινητό τηλέφωνο και ο υπολογιστής έχουν γίνει μέρος της καθημερινότητας μας. Η ραγδαία ανάπτυξη και η μαζική χρήση του internet, και κατά επέκταση των κοινωνικών δικτύων, των σελίδων ηλεκτρονικού εμπορίου κ.α από δισεκατομμύρια χρήστες έχει ως αποτέλεσμα την παραγωγή ενός τεράστιου όγκου δεδομένων, τα οποία εμπεριέχουν πολλές ωφέλιμες πληροφορίες. Ο τεράστιος όγκος και η φύση των παραπάνω δεδομένων παρουσιάζουν μεγάλες προκλήσεις, και όπως είναι φανερό η σωστή, γρήγορη και αποτελεσματική επεξεργασία τους καθίσταται άκρως αναγκαία.

Τα δεδομένα που συλλέγονται (Big Data) είναι δεδομένα δομημένα, αδόμητα και ακατέργαστα τα οποία είναι αποθηκευμένα σε διαφορετικές μορφές [1] και σε πολλαπλά μηχανήματα καθώς ο όγκος τους είναι τεράστιος και ξεπερνά τις δυνατότητες απλών αποθηκευτικών μέσων. Αυτό καθιστά τις παραδοσιακές βάσεις δεδομένων ακατάλληλες ως προς την επεξεργασία και αποθήκευση των δεδομένων. Ενδεικτικά η google το 2008 επεξεργαζόταν καθημερινά 20 PB (1 PB = 1000 GB) από δεδομένα καθημερινά. Σύμφωνα με μια έρευνα από την IDC που πραγματοποιήθηκε το 2018 [2] προβλέπεται ότι το 2025 τα ετήσια δεδομένα θα ξεπερνούν τα 175 ZB (1 ZB = 10^{12} GB).



Σχήμα 1.1: Πρόβλεψη μεγέθους παγκόσμιων δεδομένων [2]

Η χρήση μεμονωμένων μηχανημάτων για την επεξεργασία των δεδομένων, όπως συμπεραίνεται, είναι απαγορευτική και έτσι στρεφόμαστε σε κατανεμημένες μεθόδους, όπου γίνεται διαμοιρασμός των δεδομένων σε πολλαπλά μηχανήματα [3] για επεξεργασία και αποθήκευση. Τυπικό παράδειγμα θα μπορούσε να αποτελέσουν data centers μεγάλων εταιριών όπως η Google, η Amazon, το Facebook κλπ, τα οποία σε καθημερινή βάση εξυπερετούν εκατομμύρια χρήστες και επεξεργάζονται τεράστιο όγο δεδομένων.

1.1 Συστήματα διαχείρησης δεδομένων μεγάλης κλίμακας

Όπως προαναφέρθηκε, η διαχείριση αυτού του τεράστιου και ανοργάνωτου όγκου δεδομένων, δεν μπορεί να πραγματοποιηθεί με την βοήθεια κλασσικών τεχνικών. Το παραπάνω είχε ως αποτέλεσμα την δημιουργία κατανεμημένων προγραμματιστικών εργαλείων που διαμοιράζουν τα δεδομένα σε μικρά κομμάτια για παράλληλη επεξεργασία. Μερικά από τα πιο διαδεδομένα εργαλεία που χρησιμοποιούνται από την βιομηχανία είναι το Hadoop [4], το Apache Spark [3], το Flink [5], το Presto [6] κλπ. Τα παραπάνω εργαλεία έχουν αποδειχθεί αποτελεσματικά και κλιμακώσιμα και κατάλληλα για διαχείριση τεράστιου όγκου δεδομένων. Σύμφωνα με την δημοσίευση [3], υπάρχει αρχιτεκτονική που χρησιμοποιεί το apache spark που ξεπερνάει τους 8000 κόμβους.

Τα εν λόγω εργαλεία, έχουν βασιστεί, κατά ένα πολύ μεγάλο ποσοστό, στο προγραμματιστικό μοντέλο του MapReduce [7] και προσφέρουν πλήρως κατανεμημένη επεξεργασία των δεδομένων. Τα πιο σύγχρονα εργαλεία όπως το Apache Spark [3], το Flink [5] και το Presto [6] χρησιμοποιούν μια γενίκευση του MapReduce, που ονομάζονται μοντέλα κατευθυνόμενων μη-κυκλικών πεπερασμένων γράφων (Directed Acyclic Graphs - DAG) τα οποία αποτελούνται από διασωληνωμένα (pipelined) στάδια, όπου το κάθε στάδιο χωρίζεται σε κάποιες επιμέρους διεργασίες (task). Κάθε task είναι υπεύθυνο για κάποια κομμάτια δεδομένων, τα οποία ανατίθενται από τον κεντρικό διαχειριστή του cluster, στο Apache Spark ο υπεύθυνος είναι ο driver. Πέρα από το pipeline, τα εργαλεία διαθέτουν και μηχανισμούς διαχείρισης ενδιάμεσων καταστάσεων όπως είναι η διαχείριση των shuffle δεδομένων σε περίπτωση όπου θέλουμε να ανταλλάξουμε δεδομένα μεταξύ των διεργασιών.

Η διαχείριση των ενδιάμεσων δεδομένων του shuffle, αποτελεί ένα πολύ σημαντικό κομμάτι, με καθοριστικό ρόλο στην απόδοση του εκάστοτε συστήματος, καθώς η πράξη shuffle από μόνη της είναι πολύ ακριβή, και απαιτεί πλήρεις συνδέσεις μεταξύ των mapper και των reducer. Το εκάστοτε εργαλείο διαχείρισης μεγάλων δεδομένων αντιμετωπίζει το shuffling με τον δικό του τρόπο. Κατά καιρούς έχουν παρουσιαστεί δημοσιεύσεις που προσφέρουν νέους και έξυπνους τρόπους με σκοπό την βελτίωση του μηχανισμού του shuffle όπως οι δημοσιεύσεις [8], [9] κλπ.

1.2 Σκοπός

Η εργασία μας έχει ως βασικούς σκοπούς:

1. Την χρήση ενός in-memory redis cluster, ως μηχανισμό αποθήκευσης των ενδιάμεσων αποτελεσμάτων του μηχανισμού του shuffle.
2. Την χρήση της μη-σχεσιακής βάσης δεδομένων MongoDB, ως μηχανισμό αποθήκευσης των ενδιάμεσων αποτελεσμάτων του μηχανισμού του shuffle.
3. Την χρήση του kubernetes ως το βασικό εργαλείο διαχείρησης, κλιμάκωσης και deployment, για το apache spark, το redis cluster και το MongoDB.
4. Αξιολόγηση του shuffle των υλοποιήσεων μας σε σχέση με το Vanilla Spark.

1.3 Περίγραμμα

Στο πρώτο κεφάλαιο, γίνεται μια εισαγωγή στα μεγάλα δεδομένα (Big Data) και την σημασία των κατανεμημένων εργαλείων στην επεξεργασία των δεδομένων, και τον τρόπο λειτουργίας τους. Τονίζεται η σημασία του σωστού χειρισμού των ενδιάμεσων καταστάσεων, όπως το shuffling και τέλος τίθεται ο βασικός στόχος της εργασίας. Στο δεύτερο κεφάλαιο, παρουσιάζονται βασικές πληροφορίες υπόβαθρου, όπως το MapReduce, τα RDD - Resilient Distributed Datasets, ο μηχανισμός του shuffling για το εργαλείο Apache Spark και στη συνέχεια γίνεται μια μικρή εισαγωγή στο redis, το mongodb και στο kubernetes. Στο τρίτο κεφάλαιο, παρουσιάζεται η υλοποίηση της επέκτασης του μηχανισμού του shuffling του apache spark με το redis και με το Mongodb και στο τέλος γίνεται αναφορά στην μεθοδολογία δημιουργίας και κατάταξης των μερών (components) του spark, redis και mongodb στο kubernetes. Στο τέταρτο κεφάλαιο, παρουσιάζονται τα εργαλεία και η αρχιτεκτονική που ακολουθήθηκε για το deployment στο cloud, όπως και τα αποτελέσματα των δοκιμών μας. Στο πέμπτο κεφάλαιο παρουσιάζονται συνοπτικά μερικές σκέψεις για μελλοντική δουλειά πάνω στο αντικείμενο, όπως και τρόποι βελτίωσης. Τέλος στο τελευταίο κεφάλαιο παρουσιάζονται τα συμπεράσματα μας.

Κεφάλαιο 2

Υπόβαθρο

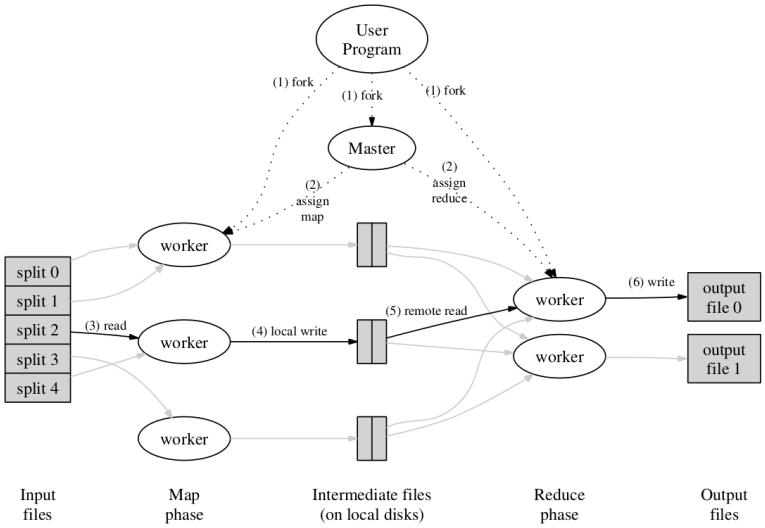
Σε αυτό το κεφάλαιο παρέχονται βασικές πληροφορίες υποβάθρου για την εργασία μας. Στο υποκεφάλαιο 2.1 βλέπουμε αναλυτικότερα την λειτουργία του Map Reduce. Στο υποκεφάλαιο 2.2 βλέπουμε αναλυτικότερα ένα από τα πιο βασικά στοιχεία του Apache Spark, το RDD (Resilient Distributed Datasets). Στο υποκεκεφάλαιο 2.3 συζητάμε σχετικά με το shuffling και την υλοποίηση του στο Apache Spark. Στο υποκεφάλαιο 2.4 γίνεται μια εισαγωγη στο Redis, έπειτα γίνεται μια εισαγωγή στο mongodb στο υποκεφάλαιο 2.5. Τέλος, στο υποκεφάλαιο 2.6 γίνεται εισαγωγή στις έννοιες του docker και kubernetes.

2.1 Map Reduce

Όπως αναφέρθηκε στο πρώτο κεφάλαιο, το MapReduce έχει παίξει ένα καθοριστικό ρόλο στον σχεδιασμό και στην λειτουργία των προαναφερθέντων εργαλείων ανάλυσης μεγάλων δεδομένων. Το MapReduce είναι ένα προγραμματιστικό μοντέλο και μια συσχετιζόμενη υλοποίηση για επεξεργασία και παραγωγή μεγάλων συνόλων δεδομένων, λειτουργεί πάνω από ένα μεγάλο cluster μηχανημάτων, είναι υψηλα κλιμακώσιμο, και ανθεκτικό στα λάθη. [7]

Το παραπάνω προγραμματιστικό μοντέλο αποτελείται απο τις 3 βασικές λειτουργίες του map, του reduce και του shuffling. Αρχικά το Map δέχεται ως είσοδο ζευγάρια από κλειδιά και τιμές και παράγει ώς έξοδο ενδίαμεσα ζευγάρια από κλειδιά-τιμές. Όλα τα ζευγάρια με το ίδιο κλειδί μεταβαίνουν στο reduce [7]. Η ενδιάμεση λειτουργία που είναι υπεύθυνση για την μετάβαση των ζευγαριών είναι το shuffling. Τέλος, το reduce δέχεται ως είσοδο το κοινό κλειδί και ένα σύνολο από τις τιμές των ζευγαριών με το κοινό κλειδί [7].

Ένας τυπικός MapReduce cluster αποτελείται από 2 βασικά είδη κόμβων τον Master και τους worker. Ο Master είναι υπεύθυνος για την ανάθεση προς εκτέλεση στους worker των map και reduce tasks. Αφού ο Master αναθέσει στον worker διεργασίες για map και reduce έπειτα του αναθέτει και ένα κομμάτι από τα δεδομένα. Ο worker διαβάζει το κομμάτι από δεδομένα που του έχει ανατεθεί, σαν είσοδο από ζευγάρια key-value, και στη συνέχεια το map task παράγει ενδιάμεσα αποτελέσματα από ζευγάρια key-value, τα οποία αποθηκεύονται στην μνήμη και περιοδικά αποθηκεύονται και στον δίσκο, σε ένα αριθμό από περιοχές όσες είναι και οι reducer (shuffling). Έπειτα, ο master ενημερώνεται για αυτές τις τοποθεσίες και με την σειρά του προωθεί αυτήν την πληροφορία στους reducer οι οποίοι διαβάζουν τα δεδομένα από τις συγκεκριμένες τοποθεσίες. Και τέλος, όπως προαναφέρθηκε, ο reducer για κάθε κλειδί σαρώνει όλα τα ενδιάμεσα δεδομένα.



Σχήμα 2.1: Επισκόπηση εκτέλεσης MapReduce [7]

Σε περίπτωση που ο master δεν λάβει περιοδική ενημέρωση για την ύπαρξη ενός worker, μέσω κάποιου μηχανισμού ping, όλα τα task τα οποία έχουν ολοκληρωθεί η βρίσκονται υπό εκτέλεση από τον worker που χάθηκε ανατίθενται από την αρχή σε έναν άλλο worker για επεξεργασία.

2.2 Resilient Distributed Dataset

Τα RDD (Resilient Distributes Datasets) είναι μια γενίκευση (abstraction) κατανεμημένης μνήμης, που επιτρέπει στους προγραμματιστές να εκτελούν υπολογισμούς εντός μνήμης σε μεγάλα cluster κρατώντας παράλληλα την ανθεκτικότητα στα λάθη και την ροή των δεδομένων όπως στο MapReduce [10], και αποτελεί έναν από τους δομικούς λίθους του εργαλείου ανάλυσης μεγάλων δεδομένων Apache Spark.

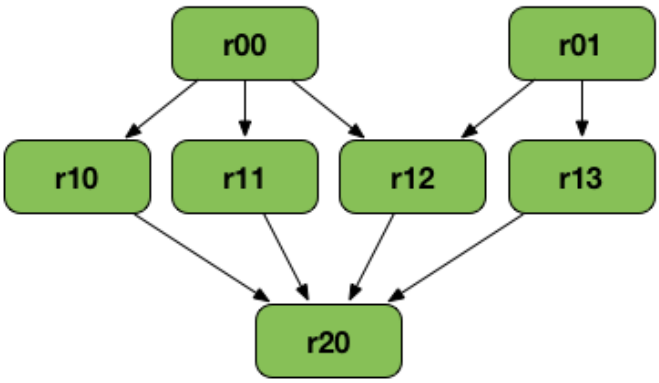
Μια από τις πιο βασικές ιδιότητες των RDDs είναι η αντοχή τους στα λάθη. Γενικότερα σε ένα σύστημα κατανεμημένης μνήμης αυτό θα μπορούσε να γίνει με κάποιο σημείο στο οποίο αποθηκεύονται τα δεδομένα (checkpointing), το οποίο είναι μη αποτελεσματικό καθώς θα έπρεπε να μεταφέρουμε μέσω δικτύου μεγάλα σύνολα δεδομένων σε όλα τα μηχανήματα, αλλά και να διαθέσουμε επιπλέον χώρο για αποθήκευση. Έτσι επιλέγεται η καταγραφή των ενημερώσεων που έχουν γίνει πάνω στα δεδομένα, αλλά και αυτό σύντομα καθίσταται μη αποδοτικό,ειδικώτερα όταν έχουμε πολλά και συχνά updates. Τα RDDs αποθηκεύουν μόνο ενημερώσεις πάνω σε ολόκληρο το σύνολο δεδομένων (coarse-grained updates), που σημαίνει ότι αποθηκεύουν τις αλλαγές μόνο μετά από κάποια μαζική αλλαγή (coarse-grained transformation), όπως είναι το map, το filter, το join κλπ. [10]

Τα RDD δεν χρειάζεται να είναι πραγματοποιημένα μέσα στην μνήμη, αντιθέτως το μόνο που χρειάζονται είναι η πληροφορία για το πως προέκυψαν από προηγούμενα RDD το οποίο αναφέρεται ως γενεαλογία (lineage) [10]. Συνήθως είναι ένας γράφος που περιέχει όλα τα RDD γονείς και τις εξαρτήσεις του . Σε περίπτωση σφάλματος,

transformation	RDD before	RDD after
$map(f : T => U)$	RDD[T]	RDD[U]
$filter(f : T => Bool)$	RDD[T]	RDD[T]
$flatMap(f : T => Seq[U])$	RDD[T]	RDD[U]
$sample(fraction : Float)$	RDD[T]	RDD[T] (Deterministic Sampling)
$groupByKey()$	RDD[(K,V)]	RDD[K,Seq[V]]
$reduceByKey(f : (V,V) => V)$	RDD[(K,V)]	RDD[(K,V)]
$union()$	(RDD[T],RDD[T])	RDD[T]
$join()$	(RDD[(K,V)],RDD[(K,W)])	RDD[(K,(V,W))]
$cogroup()$	(RDD[(K,V)],RDD[(K,W)])	RDD[(K,(Seq[V],Seq[W]))]
$crossProduct()$	(RDD[T],RDD[U]))	RDD[(T,U)]
$mapValues(f : V => W)$	RDD[(K,V)]	RDD[(K,V)] (Preserves partitioning)
$sort(c : Comparator[K])$	RDD[(K,V)]	RDD[(K,V)]
$partitionBy(p : Partitioner[K])$	RDD[(K,V)]	RDD[(K,V)]

Πίνακας 2.1: Παραδείγματα μετασχηματισμών RDD που προέρχονται από τον πίνακα 2 του paper [10]

για να επαναδημιουργηθεί το RDD θα χρειαστεί να γίνουν μετασχηματισμοί πάνω στα δεδομένα του δίσκου ή σε δεδομένα κάποιου ή κάποιων προηγούμενων RDD από τη γενεαλογία (lineage).



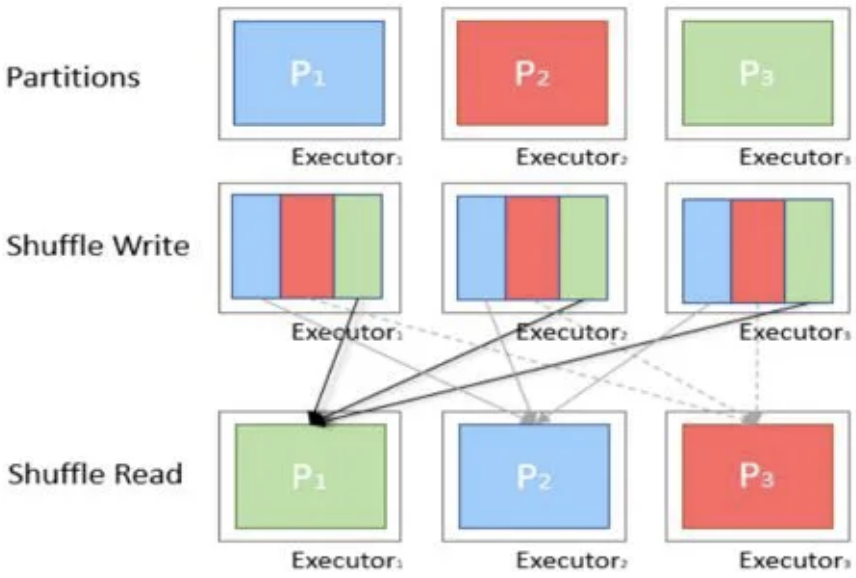
Σχήμα 2.2: Γράφος με RDD lineage [11]

2.3 Apache Spark Shuffling

Όπως αναφέρθηκε στο προηγούμενο κεφάλαιο, ο μηχανισμός shuffling είναι υπεύθυνος για την ανταλλαγή των δεδομένων των ενδιάμεσων σταδίων μεταξύ των worker.

Συγκεκριμένα θα εστιάσουμε στο Apache Spark, καθώς είναι και το εργαλείο στο οποίο έγινε η ανάπτυξη της εργασίας μας. Το Apache Spark, από προεπιλογή μας παρέχει ένα μηχανισμό shuffle ο οποίος μπορεί να είναι τοπικός είτε ως προς τον εκτελεστή (executor) της εφαρμογής σε κάθε κόμβο, είτε είναι γενικός καθολικός για όλους του εκτελεστές (executor) του worker. Αυτό σημαίνει ότι στην πρώτη περίπτωση, τα ενδιάμεσα αποτελέσματα των διεργασιών (task) της εφαρμογής (application) του συγκεκριμένου worker διαχειρίζονται από ένα τοπικό shuffle μηχανισμό, ενώ στην δεύτερη ο μηχανισμός είναι εξωτερικός και διαχειρίζεται όλα τα ενδιάμεσα αποτελέσματα των διεργασιών όλων των εφαρμογών (application) για τον συγκεκριμένο worker.

Κάθε διεργασία(task), επεξεργάζεται και ταξινομεί τα δεδομένα που της έχουν ανατεθεί και τελικά δημιουργεί, ένα shuffle αρχείο τοπικά [3] το οποίο περιέχει συγχωνευμένα και με την σειρά, όλα τα block για κάθε partition (όλα τα κλειδιά του κάθε partition έχουν το ίδια τιμή κατακερματισμού), όπως επίσης και ένα αρχείο δεικτοδότησης (index), το οποίο περιέχει την θέση του κάθε block στο αρχείο, έτσι ώστε όταν ζητηθεί ένα συγκεκριμένο block από την διεργασία του reducer να ξέρουμε την ακριβή θέση του στο αρχείο. Στην συνέχεια, η διεργασία του reduce, ενημερώνονται από τον driver για τις θέσεις των shuffle services στα οποία περιέχονται οι πληροφορίες που χρειάζονται, και στην συνέχεια ανοίγουν συνδέσεις προς όλα τα απαραίτητες υπηρεσίες shuffle όπου στέλνονται αιτήσεις για τη προσκόμιση των απαραίτητων block.



Σχήμα 2.3: Apache Spark Shuffling [12]

Η παρούσα προσέγγιση με την ταξινόμηση των δεδομένων και την αποθήκευση στον δίσκο των δεδομένων του shuffle, από την πλευρά του spark, παρέχει μια καλή ισορροπία μεταξύ απόδοσης και αντοχής σε λάθη [9] . Παρ' όλα αυτά, ο κάθε spark worker χρειάζεται σύνδεση προς όλους τους υπόλοιπους worker έτσι ώστε να προσκομίσει τα δεδομένα του shuffle. Σε περιπτώσεις όπου ο αριθμός των κόμβων είναι μεγάλος, η πιθανότητα αστοχίας ενός κόμβου είναι αρκετά μεγάλη και αυτό έχει ως συνέπεια, αν πέσει ο κόμβος, να πρέπει να επαναϋπολογίσουμε όλα τα task του κόμβου που απέτυχε.

2.4 Redis

Το Redis είναι ένα αποθηκευτικό μέσο δομών δεδομένων στην μνήμη, ανοιχτού κώδικα, που χρησιμοποιείται ως βάση δεδομένων, cache και διαμεσολαβητής μηνυμάτων. Το redis προσφέρει μια ευρεία γκάμα από δομές δεδομένων όπως strings, hashes, λίστες, σύνολα, διατεταγμένα σύνολα, ροές streams και πολλά άλλα [13].

Ως προς θέματα data persistence το redis προφέρει 2 διαφορετικές μεθόδους, η πρώτη μέθοδος, που ονομάζεται RDB (Redis Database) πραγματοποιεί στιγμιότυπα (snapshot) της κατάστασης, μετά από καθορισμένα και όχι πολύ συχνά διαστήματα και η δεύτερη μέθοδος δημιουργεί ένα αρχείο που ονομάζεται AOF (append-only-file) το οποίο καταγράφει τις αλλαγές, θεωρείται πιο ανθεκτικό σε σχέση με το rdb αλλά είναι πολύ μεγαλύτερο σε μέγεθος. [14]

Το redis παρέχει ενσωματωμένο μηχανισμό αντιγραφής (replication) των δεδομένων, το οποίο το συναντάμε περισσότερο όταν το redis έχει ρυθμιστεί ως cluster. Σε αυτήν την περίπτωση, ασύγχρονα ο κάθε master παράγει ένα αριθμό από replicas (ο οποίος έχει οριστεί από εμάς) των δεδομένων του, τα οποία αποθηκεύονται σε κόμβους που φυλάνε αντίγραφα.

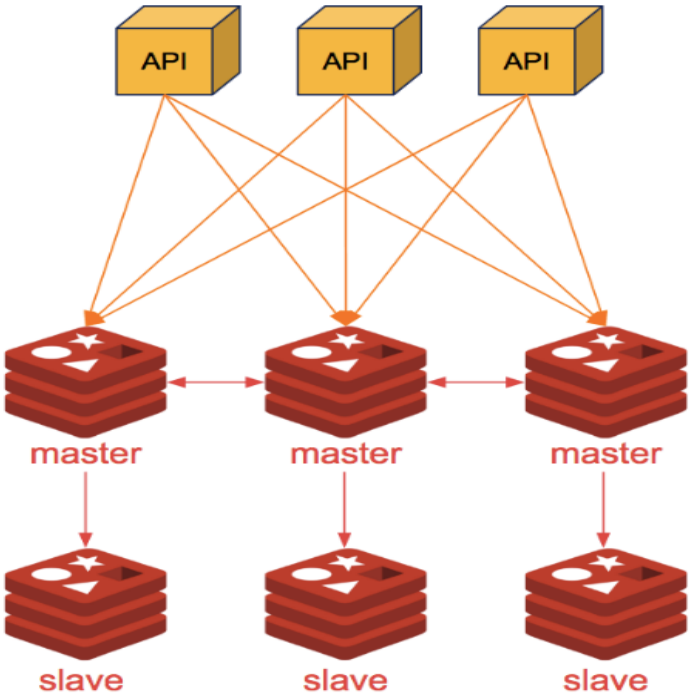
Η πρόσβαση στο redis γίνεται μέσω κάποιου client, είτε μέσω κάποιου τερματικού (terminal) με το redis-cli, είτε μέσω κάποιου προγράμματος με την βοήθεια βιβλιοθηκών. Για παράδειγμα 2 βιβλιοθήκες είναι το jedis και το lettuce. Ενδεικτικά κάποιες από τις πιο βασικές εντολές που εκτελούμε στο redis είναι το GET,SET,DEL κλπ, όπου το get μας επιστρέφει την τιμή του κλειδιού που αναζητάμε, το set αναθέτει σε ένα κλειδί την συγκεκριμένη τιμή που του θέτουμε και το del διαγράφει το αναφερόμενο κλειδί.

Το redis αποτελεί ένα από τα βασικότερα κομμάτια της εργασίας και χρησιμοποιείται ως το μέσο αποθήκευσης των δεδομένων για το ενδιάμεσο στάδιο του shuffling, όπου κάθε block αποθηκεύεται ως μια εγγραφή κλειδιού τιμής (key-value). Πιο συγκεκριμένα στην εφαρμογή μας χρησιμοποιείται η κατανεμημένη μορφή του Redis, που ονομάζεται Redis Cluster, λόγω της κλιμακωσιμότητας του εν λόγω προγράμματος. Πιο συγκεκριμένα θα αναφερθούμε στο Redis Cluster στο αμέσως επόμενο υποκεφάλαιο.

2.4.1 Redis Cluster

Όπως προαναφέρθηκε το redis cluster είναι η βασική αρχιτεκτονική που χρησιμοποιούμε για την αποθήκευση των shuffle δεδομένων. Στην συγκεκριμένη αρχιτεκτονική, τα δεδομένα είναι διαμοιρασμένα μεταξύ πολλαπλών κόμβων, όπου η επιλογή του κόμβου αποθήκευσης μιας συγκεκριμένης εγγραφής γίνεται αυτόματα. Παράλληλα το clutering προσφέρει και ένα βαθμό διαθεσιμότητας καθώς δεν σταματάει η λειτουργία σε περίπτωση αστοχίας κόμβων.

Το redis αποτελείται από 16384 θέσεις(slots) τα οποία μοιράζονται στους κόμβους του cluster ανάλογα με το πλήθος τους. Η επιλογή της θέσης (slot) και συνεπώς του κόμβου αποθήκευσης μιας εγγραφής γίνεται αυτόματα και πιο συγκεκριμένα μέσω χρήσης μιας συνάρτησης κατακερματισμού. Με αυτόν τον τρόπο, επιτυγχάνεται πολύ



Σχήμα 2.4: Redis Cluster [15]

υψηλή κλιμάκωση, ειδικότερα όταν ο αριθμός των κόμβων είναι μεγάλος. Σε περίπτωση που ένα κόμβος παρουσιάσει κάποιου είδους αστοχία ή δεν μπορεί να υπάρξει σύνδεση με τους υπόλοιπους κόμβους, για να μην χαθούν τα δεδομένα που βρίσκονται στις θέσεις που έχει αναλάβει ο συγκεκριμένος κόμβος, η αρχιτεκτονική του Redis Cluster χρησιμοποιεί ένα μοντέλο master-replica. Όπου κάθε θέση (slot) έχει και ένα αριθμό από αντίγραφα N, τον οποίο ορίζει ο δημιουργός του cluster. Αυτό πραγματοποιείται, δημιουργώντας και κάποιους κόμβους που λειτουργούν μόνο σαν αντίγραφα όπου κάθε αντίγραφο αντιστοιχίζεται σε έναν master έτσι ώστε σε περίπτωση αστοχίας ενός master ο replica κόμβος να αναβαθμιστεί σε master [16].

2.5 MongoDB

Το Mongodb αποτελεί μια βάση δεδομένων εγγράφων (document database) ανοιχτού κώδικα, όπου ένα έγγραφό (document) είναι ένα μητρώο (record) το οποίο αποθηκεύει πληροφορίες σε ζευγάρια key-value και οι τιμές (values) μπορεί να αποτυπώσουν μια ευρεία γκάμα από τύπους δεδομένων και δομών όπως strings, αριθμούς, ημερομηνίες, πίνακες, αντικείμενα κλπ. Ένα έγγραφο (document) μπορεί να αποθηκευτεί σε μορφή JSON, BSON και XML. Το μοντέλο της βάσης δεδομένων εγγράφων, προσφέρει πολλά πλεονεκτήματα, είναι γρήγορο και εύκολο ως προς την εργασία, ευέλικτο καθώς ένα μοντέλο εξελίσσεται παράλληλα με την εφαρμογή και καθώς δεν υπάρχει κάποιο σταθερό μοντέλο το οποίο πρέπει να ικανοποιείται από τα δεδομένα, αυτό τις καθιστά μη σχεσιακές (NoSQL)[17]. Τέλος, το mongodb μας παρέχει την δυνατότητα της οριζόντιας κλιμάκωσης (horizontal scaling) και πετυχαίνει υψηλή

διαθεσιμότητα.

Μια από τις πιο βασικές έννοιες του MongoDB αποτελούν οι συλλογές (collections). Όπως υποδηλώνει και το όνομα μια συλλογή αποτελεί μια ομάδα από έγγραφα, τα οποία σε γενικές γραμμές αποθηκεύουν παρόμοια περιεχόμενα[17]. Τα έγγραφα δεν χρειάζεται απαραίτητα να διαθέτουν τα ίδια πεδία όπως π.χ. σε μια σχεσιακή βάση δεδομένων, καθώς όπως προαναφέρθηκε μας παρέχουν ευελιξία. Παρ' όλα αυτά, υπάρχει και η δυνατότητα της επικύρωσης σχήματος (schema validation), με βάση κάποιους κανόνες έτσι ώστε οι εισαγωγές και οι ενημερώσεις να έχουν ένα προκαθορισμένο σχήμα, σε περίπτωση που αυτό είναι αναγκαίο.

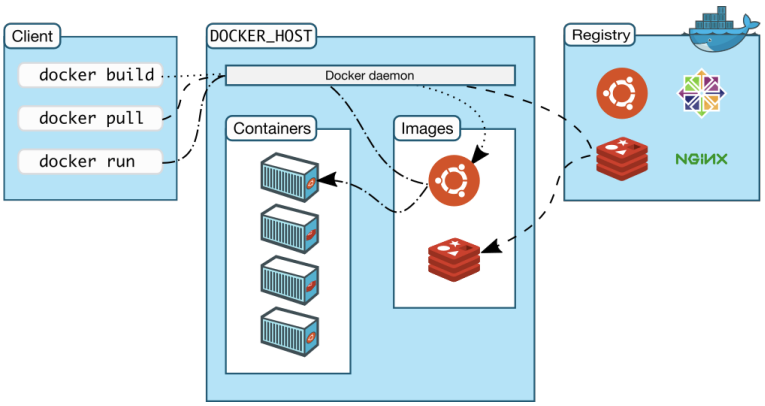
Υπάρχουν διάφορες στρατηγικές παράταξης (deployment) του MongoDB όπως το ReplicaSet το οποίο αποτελείται από μια ομάδα από MongoDB διεργασίες, οι οποίες αναλαμβάνουν να διαχειριστούν και να συντηρήσουν τα ίδια δεδομένα. Ένα ReplicaSet μας προφέρει υψηλή διαθεσιμότητα και πλεονασμό[18]. Ένα ReplicaSet αποτελείται από πολλαπλούς κόμβους οι οποίοι συντηρούν τα δεδομένα και εναλλακτικά μπορεί να υπάρχει ένας διαιτητής (arbiter) κόμβος ο οποίος συμμετέχει σε λειτουργίες όπως ψηφοφορίες. Ένας από τους κόμβους που διατηρούν δεδομένα εκλέγεται ως ο πρωταρχικός (primary), και πάντα πρέπει να είναι μόνο ένας, οι υπόλοιποι ορίζονται ως δευτερεύοντες (secondary). Ο πρωταρχικός κόμβος λαμβάνει όλες τις λειτουργίες εγγραφής και καταγράφει τις αλλαγές που πραγματοποίησε σε ένα ημερολόγιο (log). Οι δευτερεύοντες κόμβοι αναλαμβάνουν να εκτελέσουν τις αλλαγές που πραγματοποίησε ο πρωταρχικός κόμβος στο ημερολόγιο (log). Σε περίπτωση αστοχία πρωταρχικού (primary) κόμβου, πραγματοποιείται πάλι ψηφοφορία για αναβάθμιση δευτερεύοντα (secondary) κόμβου σε πρωταρχικό[18].

Μια άλλη στρατηγική, η οποία χρησιμοποιήθηκε και στην υλοποίηση μας, είναι η στρατηγική του sharded cluster, η οποία αποτελείται από την συνιστώσα του shard, όπου κάθε shard αποτελεί και ένα ReplicaSet το οποίο περιέχει ένα υποσύνολο των δεδομένων. Επιπλέον το sharded cluster διαθέτει την συνιστώσα του mongos το οποίο λειτουργεί ως ένας δρομολογητής ερωτημάτων και αποτελεί την διεπαφή επικοινωνίας της βάσης με τους client. Η τελευταία διεπαφή είναι οι διακομιστές ρυθμίσεων (config server), ο οποίος αποθηκεύει μεταδεδομένα και ρυθμίσεις για το cluster και αυτός όπως και τα shard πρέπει να είναι σε μορφή ReplicaSet.

2.6 Docker και Kubernetes

Το docker είναι μια πλατφόρμα για ανάπτυξη, φόρτωση και εκτέλεση εφαρμογών, η οποία επιτρέπει την διαχώριση της εφαρμογής από την υποδομή. Το docker μας προσφέρει την ικανότητα της πακετοποίησης και της εκτέλεσης της εφαρμογής μας σε ένα ελαφρώς απομονωμένο περιβάλλον που ονομάζεται container. Τα container περιέχουν όλα όσα χρειάζεται η εφαρμογή για να τρέξει και έτσι δεν χρειάζεται να εγκαταστήσουμε τοπικά απαραίτητα στοιχεία για την εκτέλεση της εφαρμογής [19], τα container δημιουργούνται μέσα από ένα πρότυπο, το οποίο ονομάζεται εικόνα (image) και περιέχει ένα σύνολο από εντολές οι οποίες όταν τρέξουν δημιουργούν ένα container.

Το docker αποτελεί μια πολύ καλή λύση καθώς μας επιτρέπει την δημιουργία



Σχήμα 2.5: Αρχιτεκτονική Docker [19]

φορητών εργασιών (workload), τα οποία μπορούν να τρέξουν σε μια ευρεία γκάμα από περιβάλλοντα όπως σε ένα φυσικό κόμβο, σε ένα εικονικό (virtual) κόμβο, στο cloud ακόμα και σε ένα συνδυασμό από αυτά. Με αυτόν τον τρόπο μπορούμε εύκολα να δημιουργήσουμε εφαρμογές οι οποίες κλιμακώνουν εύκολα αλλά και αυτό δημιουργεί κάποιες δυσκολίες, καθώς η διαχείριση πολλών container μπορεί να αποτελέσει μια περίπλοκη διαδικασία.

Ένα εργαλείο το οποίο διαχειρίζεται αυτόματα την επανάκαμψη από λάθη (failover) και την κλιμάκωση των container ονομάζεται Kubernetes, και μας παρέχει κάποιες πολύ χρήσιμες υπηρεσίες. Για παράδειγμα δύο πολύ χρήσιμες υπηρεσίες είναι η αυτόματη ανάρρωση (self-healing) και η αυτόματη επιλογή καλαθιού (automatic bin picking). Το self-healing μπορεί να επανεκκινήσει container σε περίπτωση που αποτύχουν, μπορεί να αντικαταστήσει και να "σκοτώσει" (kill) container ανάλογα με ορισμένους ελέγχους υγείας (health-check) που πραγματοποιούνται. Με το automatic bin picking μπορούμε να επιλέξουμε σε ποιον κόμβο μπορούμε να τρέξουμε το container μας. Παράλληλα, στο kubernetes μπορούμε να επιλέξουμε και τους πόρους του κάθε container ορίζοντας τους (CPU και RAM) που θα καταναλώσει ο κάθε container[20].

Κεφάλαιο 3

Αρχιτεκτονική και υλοποίηση

Στο παρόν κεφάλαιο παρουσιάζεται η αρχιτεκτονική και η υλοποίηση της επέκτασης κομματιών του shuffle service με γνωστές κατανεμημένες βάσεις δεδομένων. Πιο συγκεκριμένα, στο πρώτο υποκεφάλαιο 3.1 θα παρουσιαστεί η επέκταση με το Redis. Στο δεύτερο υποκεφάλαιο 3.2 θα παρουσιαστεί η επέκταση με το mongodb. Στην συνέχεια στο τρίτο υποκεφάλαιο 3.3 παρουσιάζουμε συνοπτικά την μεθοδολογία πακετοποίησης (containarization) του Spark, του redis και του mongo και τον τρόπο παράταξης (deployment) στο kubernetes.

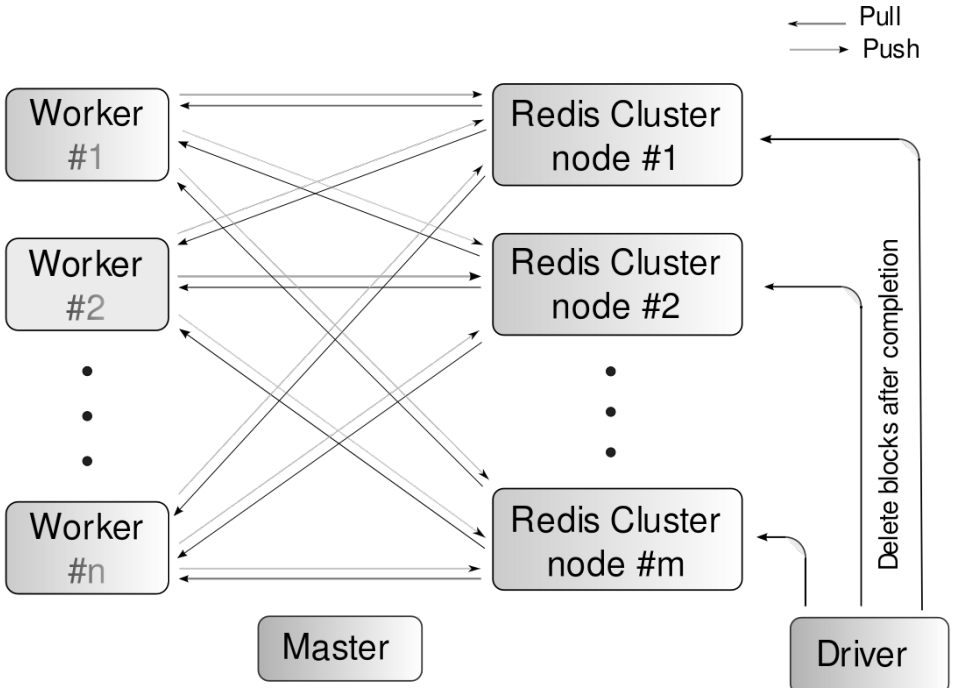
3.1 Επέκταση του shuffle service με in-memory Redis

Στο υποκεφάλαιο αυτό, θα παρουσιάσουμε την επέκταση του shuffle service με την βοήθεια του in-memory Redis, το οποίο σημαίνει πως τα ενδιάμεσα δεδομένα αποθηκεύονται πάνω στην μνήμη (RAM). Σε αυτό το σημείο αξίζει να σημειωθεί πως η παρούσα υλοποίηση αποτελεί ένα proof-of-concept και έχει εξεταστεί με το stress test workload που παρουσιάζεται στο υποκεφάλαιο 4.3. Επίσης πρέπει να σημειωθεί πως υποθέτουμε ότι κατά την διάρκεια της εκτέλεσης δεν χάνονται κόμβοι και κατά συνέπεια δεν έχουμε απώλεια ενδιάμεσων δεδομένων κατά την διάρκεια του shuffle, από την πλευρά της βάσης. Αυτό θα είχε ως αποτέλεσμα να πρέπει να διαχειριστούμε fetch fail exception, τα οποία παράγει το spark σε περίπτωση που δεν μπορεί να διαβάσει τα δεδομένα κάποιου block.

3.1.1 Λειτουγία του shuffle με το Redis

Όπως φαίνεται στο σχήμα 3.1 το redis έχει διαμορφωθεί σε κατανεμημένη αρχιτεκτονική, και αναλαμβάνει την αποθήκευση των ενδιάμεσων αποτελεσμάτων της λειτουργίας του shuffle. Για τον σκοπό αυτό οι worker του spark πρέπει να μπορούν να επικοινωνούν με το redis, έτσι ώστε να φορτώσουν τα ενδιάμεσα block απ' αυτό, αλλά και να μπορούν να διαβάσουν από αυτό τα block τα οποία χρειάζονται. Με αυτόν τον τρόπο οι worker ακολουθούν μια αρχιτεκτονική απόπλεξης (disaggregated), με την οποία απελευθερώνονται από την φύλαξη και παροχή των ενδιάμεσων αποτελεσμάτων. Αυτό σημαίνει ότι η παρουσία των worker είναι εφήμερη, καθώς η ύπαρξη τους δεν είναι απαραίτητη για την άντληση των αναγκαίων ενδιάμεσων αποτελεσμάτων. Κατά την λήξη της εφαρμογής του spark, την διαγραφή των ενδιάμεσων αποτελεσμάτων, από την μνήμη, αναλαμβάνει ο driver, ο οποίος διαγράφει τα block, όπως φαίνεται και από το σχήμα 3.1.

Στην παρούσα εργασία το redis χρησιμοποιείται σε μορφή cluster, που σημαίνει ότι



Σχήμα 3.1: Spark - Redis Block Διάγραμμα

θα υπάρχει διαμοιρασμός των ενδιάμεσων δεδομένων μεταξύ των κόμβων του Redis. Πιο συγκεκριμένα, το redis αποτελείται από slots ή buckets με συνολικό πλήθος 16384. Ώστε, όταν επιλέγεται ως αρχιτεκτονική το cluster, τα slots μοιράζονται μεταξύ των master κόμβων του redis.

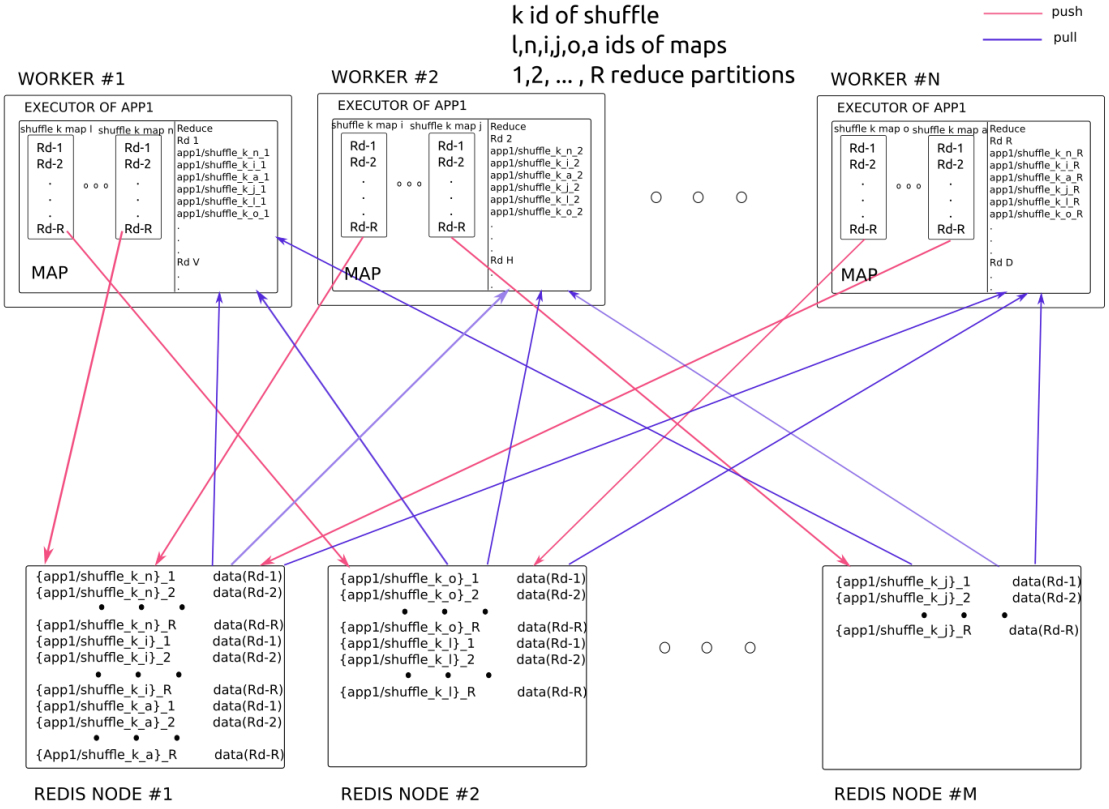
Για παράδειγμα σε μια αρχιτεκτονική με 6 master ένας πιθανός διαμοιραζμός των θέσεων (slot) είναι ο εξής:

1. M1 slots [0-2730]
2. M2 slots [2731-5460]
3. M3 slots [5461-8191]
4. M4 slots [8192-10922]
5. M5 slots [10923-13652]
6. M6 slots [13653-16383]

Η επιλογή της θέσης (slot) και εν συνεχεία του κόμβου για ένα κλειδί γίνεται τυχαία μέσω μιας μεθόδου κατακερματισμού, έτσι ώστε να υπάρξει ομοιόμορφος διαμοιρασμός των δεδομένων στους κόμβους. Πιο συγκεκριμένα: το redis χρησιμοποιεί την μέθοδο κατακερματισμού CRC16, οπότε η επιλογή του slot δίνεται με τον παρακάτω τύπο :

$$CRC16(key) \mod 16384 = SlotOfCurrentKey$$

Στην περίπτωση του shuffle, η κάθε map διεργασία παράγει και ένα ενδιάμεσο αρχείο shuffle, το οποίο περιέχει όλα τα δεδομένα των block (reduce partition) τα οποία



Σχήμα 3.2: Spark - Redis ροή δεδομένων

βρίσκονται μετατοπισμένα εντός αυτού. Όπως προαναφέραμε και στο κεφάλαιο 2.3. Αυτό έχει όνομα $shuffle\_ShuffleId\_MapId\_0$, όπου shuffleId, όπως υποδηλώνει και το όνομα, είναι το μοναδικό αναγνωριστικό του συγκεκριμένου shuffle, το MapId είναι το αναγνωριστικό του συγκεκριμένου Map, άρα κάθε block που βρίσκεται στο συγκεκριμένο αρχείο έχει όνομα $shuffle\_ShuffleId\_MapId\_ReduceId$, όπου ReduceId είναι το μοναδικό αναγνωριστικό ενός συγκεκριμένου reduce partition όπως φαίνεται και στο σχήμα 3.2.

Στην υλοποίηση μας θεωρούμε πως ένα key - value ζευγάρι για το redis, αποτελεί ένα shuffle block όπως αναφέραμε παραπάνω. Όπου το κλειδί αποτελεί :

$$KeyGen(appId, shuffle\_ShuffleId\_MapId\_ReduceId) =$$

$$\{appId/shuffle\_ShuffleId\_MapId\}\_ReduceId$$

και η τιμή είναι τα δεδομένα του block. Ο λόγος που το string βρίσκεται εν μέρει μέσα σε αγκύλες είναι: επειδή το Redis θα περάσει μέσα από την μέθοδο κατακερματισμού μόνο ότι βρίσκεται εντός των αγκυλών δηλαδή:

$$RedisHash(\{appId/shuffle\_ShuffleId\_MapId\}\_ReduceId) =$$

$$RedisHash(appId/shuffle\_ShuffleId\_MapId)$$

αυτό θα έχει ως αποτέλεσμα όλα τα block που ανήκουν στο ίδιο shuffle , map αλλά και application να καταχωρηθούν στον ίδιο κόμβο.

Αυτό μπορεί να έχει πλεονεκτήματα, σε μελλοντικές υλοποιήσεις, στην περίπτωση πτώσης ενός κόμβου, ώστε να μπορέσουμε να προσδιορίσουμε ακριβώς ποια shuffle αρχεία χάθηκαν. Π.χ. σε περίπτωση που κατακερματίζαμε ολόκληρο το αναγνωριστικό του block, θα σήμαινε πως blocks του ίδιου map θα πήγαιναν τυχαία σε διαφορετικούς κόμβους και θα είχε ως αποτέλεσμα, σε περίπτωση που πέσει ένας κόμβος, να χάσουμε blocks από περισσότερα maps. Αυτό θα είχε μεγαλύτερο κόστος στον επαναϋπολογισμό, καθώς θα έπρεπε να επαναϋπολογιστούν από την αρχή περισσότερες διεργασίες (task). Επίσης, ένα άλλο πλεονέκτημα, που μπορεί να έχει, είναι και στο διάβασμα, καθώς οι reducer έχουν μεγαλύτερη πιθανότητα να διαβάζουν πιο ομοιόμορφα, υποθέτοντας πως τα maps έχουν διαμοιραστεί ομοιόμορφα στους redis κόμβους.

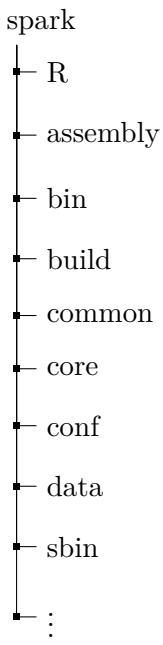
Ο λόγος που το μοναδικό αναγνωριστικό (id) της εφαρμογής εμπεριέχεται στον υπολογισμό του κλειδιού είναι για να δημιουργήσουμε μια μη ντετερμινιστικότητα στην επιλογή του κόμβου, καθώς αν δεν υπήρχε το id του application, το μοναδικό αναγνωριστικό του block $shuffle\_ShuffleId\_MapId$ θα είχε πάντα την ίδια έξοδο, από την μέθοδο κατακερματισμού και άρα η επιλογή του κόμβου θα ήταν πάντα η ίδια. Άλλο ένα πλεονέκτημα που μας προσφέρει το μοναδικό αναγνωριστικό (id) στην εφαρμογή είναι ότι μαρκάρει όλα τα block προς διαγραφή, ώστε, όταν έρθει σε πέρας η εφαρμογή μας, ο driver να ξέρει ποια blocks να διαγράψει.

Άρα εν κατακλείδα, η ροή των δεδομένων από τους workers του spark, από και προς το redis cluster, όπως φαίνται στο σχήμα, 3.2 μπορεί να αποτυπωθεί με τα εξής βασικά σημεία :

1. Αρχικά οι διεργασίες του map (map task) στον worker επεξεργάζονται τα δεδομένα και δημιουργούν shuffle αρχεία
2. Διαβάζεται το κάθε block απο κάθε αρχείο και αποθηκεύται ως key - value από το redis
3. Ο κόμβος επιλέγεται με βάση του αποτελέσματα της μεθόδου κατακερματισμού με το κλειδί του ζευγαριού
4. Το κλειδί είναι ειδικά διαμορφώμένο έτσι ώστε όλα τα blocks του ίδιου shuffle file να αποθηκευτούν στον ίδιο κόμβο για τους λόγους που προαναφέρθηκαν
5. Τέλος οι διεργασίες reduce (reduce task) διαβάζουν, τα block που χρειάζεται από το redis cluster αντι για τους worker

3.1.2 Κώδικας Apache Spark

Η επέκταση του shuffle service του apache spark (στην εφαρμογή μας χρησιμοποιήθηκε η έκδοση 3.2.0) με το redis έχει ως αποτέλεσμα την τροποίηση - διαμόρφωση των πηγαίων αρχείων του spark και την δημιουργία νέων αρχείων πηγαίου κώδικα. Αρχικά, το πρώτο βήμα είναι η πρόσβαση στον πηγαίο κώδικα του spark από το GitHub που βρίσκεται στο URL [21], και το δένδρο φακέλων (folder tree) έχει την παρακάτω μορφη (ενδεικτικά μερικοί από τους φάκελους) :



Ενδεικτικός ρόλος μερικών κρίσιμων φακέλων για την εφαρμογή μας :

1. Ο φάκελος conf περιέχει τις ρυθμίσεις (configuration) του apache spark για συκεκριμένες λειτουργίες. Ενδεικτικά: Το log4j.properties που μας παρέχει ρυθμίσεις (configuration) για τα logs που παράγει το Apache Spark, το αρχείο metrics.properties που μας παρέχει ρυθμίσεις για το εσωτερικό σύστημα ματρικών του spark τα αρχεία spark-defaults.conf και spark-env.sh, που θέτουν προκαθορισμένες μεταβλητές περιβάλλοντος(default environment variables) , όπως είναι για παράδειγμα ο SPARK_MASTER_HOST που δεσμεύει τον master σε μια διαφορετική IP.

2. Ο φάκελος build που περιέχει script του Maven [22] και του SBT, [23] για το χτίσιμο και την μεταγλώττιση(compile) του project του apache spark.

3. Ο φάκελος sbin περιέχει script για έναρξη δομικών στοιχείων (component), πιο συγκεκριμένα σε αυτήν την εργασία χρησιμοποιούνται τα script start-master.sh και start-worker.sh για την έναρξη του master και του worker.

4. Τέλος, ο φάκελος core, που όπως υποδεικνύει και το όνομα του, περιέχει τα βασικότερα αρχεία πηγαίου κώδικα. Περιέχει τις κλάσεις που προσφέρουν τις βασικότερες λειτουργίες του spark, όπως για παράδειγμα : η κλάση του Master (package path: org.apache.spark.deploy.master.Master), η κλάση του worker (package path: org.apache.spark.deploy.worker.Worker), η κλάση του BlockManager (package path: org.apache.spark.storage.BlockManager) κλπ. Σε αυτό το κεφάλαιο θα εστιάσουμε σε αυτό τον φάκελο, όπου θα διαμορφώσουμε και θα δημιουργήσουμε αρχείου πηγαίου κώδικα.

3.1.3 Υλοποίηση του shuffle με το Redis στο Apache Spark

Παρακάτω παραθέτουμε τα αρχεία (κλάσεις) του Apache Spark που βρίσκονται στον φάκελο (core) όπως προαναφέρθηκε στο υποκεφάλαιο 3.1.2, και την τοποθεσία τους (package path τους και το σχετικό μονοπάτι (relative path) σε σχέση με τον φάκελο core), τα οποία πραγματοποιούν τις αναγκαίες λειτουργίες και πρέπει σε κάποιο βαθμό να τροποποιηθούν για την επέκταση του Apache Spark:

Κλάση	Package path	Relative path from
Abstract Class		
$ShuffleWriter[K, V]$	org.apache.spark.shuffle .ShuffleWriter	core/src/main/scala/org/ apache/spark/shuffle/ ShuffleWriter.scala
Inherited Classes		
$BypassMergeSortShuffleWriter[K, V]$	org.apache.spark.shuffle.sort .BypassMergeSortShuffleWriter	core/src/main/java/org/ apache/spark/shuffle/sort/ BypassMergeSortShuffleWriter.java
$UnsafeShuffleWriter[K, V]$	org.apache.spark.shuffle.sort .UnsafeShuffleWriter	core/src/main/java/org/ apache/spark/shuffle/ sort/UnsafeShuffleWriter.java
$SortShuffleWriter[K, V, C]$	org.apache.spark.shuffle.sort .SortShuffleWriter	core/src/main/scala/org/ apache/spark/shuffle/ sort/SortShuffleWriter.scala
$IndexShuffleBlockResolver$	org.apache.spark.shuffle .IndexShuffleBlockResolver	core/src/main/scala/org/ apache/spark/shuffle/ IndexShuffleBlockResolver.scala
$BlockManager$	org.apache.spark.storage .BlockManager	core/src/main/scala/org/ apache/spark/storage/ BlockManager.scala
$ShuffleBlockFetcherIterator$	org.apache.spark.storage .ShuffleBlockFetcherIterator	core/src/main/scala/org/ apache/spark/storage/ ShuffleBlockFetcherIterator.scala
$StandaloneAppClient$	org.apache.spark.deploy.client .StandaloneAppClient	core/main/scala/org/ apache/spark/deploy/client/ StandaloneAppClient.scala
package $config$	org.apache.spark.internal .config.package	core/main/scala/org/ apache/spark/internal/config/ package.scala

Πίνακας 3.1: Πίνακας χρήσημων κλάσεων Apache Spark

Η κλάση ShuffleWriter 3.1 αποτελεί την βασική γενικευμένη (abstract) κλάση που χρησιμοποιείται όταν θέλει μια διεργασία map (map task) να γράψει τα δεδομένα της, που βρίσκονται σε μορφή κλειδιού τιμής (key-value), σε ένα αρχείο shuffle. Ανάλογα με τον αριθμό των partition των δεδομένων, και του ζητουμένου RDD (Resilient Distributed Dataset), επιλέγεται μια βελτιστοποιημένη κλάση Shuffle Write, ως προς το πρόβλημα που κληρονομεί, την γενικευμένη (abstract) κλάση ShuffleWriter. Όπως φαίνεται και στον πίνακα 3.1 αυτές οι κλάσεις είναι: BypassMergeSortShuffleWrite,

UnsafeShuffleWriter, SortShuffleWriter.

Η κλάση IndexShuffleBlockResolver περιέχει την μέθοδο writeMetadataFileAndCommit που είναι υπεύθυνη για το τελική υποβολή (commit) των αρχείων του shuffle, όπως και των μεταδεδομένων (metadata) αρχείων του shuffle (index, checksum) στον δίσκο, αφότου ο Shuffle Write έχει τελειώσει την εγγραφή των δεδομένων. Ο BlockManager αποτελεί το βασικό διαχειριστή των ενδιάμεσων αρχείων του shuffle και είναι αυτό που αναλαμβάνει να διαχειριστεί αιτήματα για προσκόμιση blocks από τον δίσκο.

Η κλάση ShuffleBlockFetcherIterator αποτελεί την βασική κλάση μιας reduce διεργασίας (reduce task), για την προσκόμιση των απαραίτητων block για την λειτουργία και είναι υπεύθυνη για την δημιουργία FetchRequest προς τους υπόλοιπους worker, έτσι ώστε να μεταφερθούν τα blocks. Η κλάση StandaloneAppClient επιτρέπει σε μια εφαρμογή να επικοινωνεί με τον διαχειριστή του cluster (Cluster Manager) και ενημερώνεται για κάποια γεγονότα όπως π.χ. πότε τελειώνει το Spark Application. Τέλος, το package config περιέχει μεταβλητές περιβάλλοντος (environment variables) που μπορεί να είναι προκαθορισμένες (default), είτε να έχουν ρυθμιστεί από τα conf αρχεία που αναφέραμε στο υποκεφάλαιο,3.1.2 είτε να ρυθμίζονται από τα ορίσματα (arguments) που δίνει ο χρήστης.

Στην συνέχεια παραθέτουμε τα αρχεία πηγαίου κώδικα που δημιουργήθηκαν για την επέκταση του shuffle με το RedisCluster μας:

Κλάση	Package path	Relative path from
RedisServerEnv	org.apache.spark .RedisServerEnv	core/src/main/scala/org/ apache/spark/ RedisServerEnv.scala
RedisRemoveBlocks	org.apache.spark.shuffle.redis .RedisRemoveBlocks	core/src/main/java/org/ apache/spark/shuffle/ redis/RedisRemoveBlocks.java
RedisUpload	org.apache.spark.shuffle.redis .RedisUpload	core/src/main/scala/org/ apache/spark/shuffle/ redis/RedisUpload.java

Πίνακας 3.2: Πίνακας χρήσημων κλάσεων Redis Apache Spark

Η κλάση RedisServerEnv εμπεριέχει τον client με τον οποίο επικοινωνούμε με το Redis Cluster μας. Η κλάση RedisRemoveBlocks, όπως υποδηλώνει και το όνομα της, χρησιμοποιείται από τον Driver έτσι ώστε, μετά το πέρας της εφαρμογής, να διαγράψει τα block από το Redis Cluster. Τέλος, η RedisUpload χρησιμοποιείται για την μεταφόρτωση των shuffle block στο Redis Cluster.

Για την επικοινωνία με το Redis Cluster μας, όπως προαναφέραμε, χρησιμοποιούμε ένα client ο οποίος διαχειρίζεται την επικοινωνία μας με το Cluster, δημιουργώντας έτσι ένα επίπεδο αφαιρετικότητας μεταξύ του χρήστη και της διαχείρισης της επικοινωνίας του cluster. Στην παρούσα εργασία χρησιμοποιήθηκε το Jedis, το οποίο είναι ένα client για επικοινωνία μιας εφαρμογής Java με κάποια Redis αρχιτεκτονική. Είναι σχεδιασμένος για απόδοση αλλά και για ευκολία χρήσης [24]. Στην εργασία μας θα εστιάσουμε στο JedisCluster το οποίο μας επιτρέπει την σύνδεση σε Redis Cluster.

Το package config, όπως έχουμε προαναφέρει περιέχει μεταβλητές περιβάλλοντος (environment variables) που ρυθμίζονται με διάφορους τρόπους. Σε αυτό το σημείο προστέθηκαν και μερικές custom μεταβλητές, έτσι ώστε να μπορέσουμε να ενημερώσουμε το spark από την γραμμή εντολών (command line) του spark-submit ώστε να χρησιμοποιήσει το redis ως κύριο μηχανισμό για το shuffle, αλλά και να ρυθμίσουμε το client στα μέτρα μας.

Environment Μεταβλητή	Command line όνομα	default τιμή	Χρήση
SPARK_REDIS_SHUFFLE_ENABLE	spark.redis.shuffle.enable	False	Ενεργοποίηση του shuffle με redis
SPARK_REDIS_SHUFFLE_CLUSTER_IP	spark.redis.shuffle.cluster.ip	127.0.0.1	IP ενός Redis Master οι υπόλοιποι βρίσκονται αυτόματα από τον client
SPARK_REDIS_SHUFFLE_PORT	spark.redis.shuffle.port	7000	Η πόρτα του Redis Master
SPARK_REDIS_MAX_REQS_PER_EXECUTOR	spark.redis.shuffle .max.reqs.per.executor	10	Αριθμός συνδέσεων που ανοίγει ο client
SPARK_REDIS_SETTINGS_ENABLE_MIN_IDLE	spark.redis.settings .enable.min.idle	false	Μεταβλητή που ορίζει το ελάχιστο αριθμό αδρανών συνδέσεων, false = 1 min idle, true = maxIndle = total connections
SPARK_AGGREGATED_FETCH	spark.aggregated .fetch	false	ενεργοποιεί το pipeline για την ανάγνωση των δεδομένων

Πίνακας 3.3: Πίνακας environment μεταβλητών Redis

Παράδειγμα εκτέλεσης spark - submit για ενεργοποίηση του Redis

$$spark - submit\ master : //master\_ip : master\_port$$

$$- - conf\ spark.redis.shuffle.enable = true$$

$$- - conf\ spark.redis.shuffle.cluster.ip = RedisMasterIP$$

$$- - conf\ spark.redis.shuffle.port = RedisMasterPort$$

$$--conf\ spark.redis.shuffle.max.reqs.per.executor = ConnectionNum$$

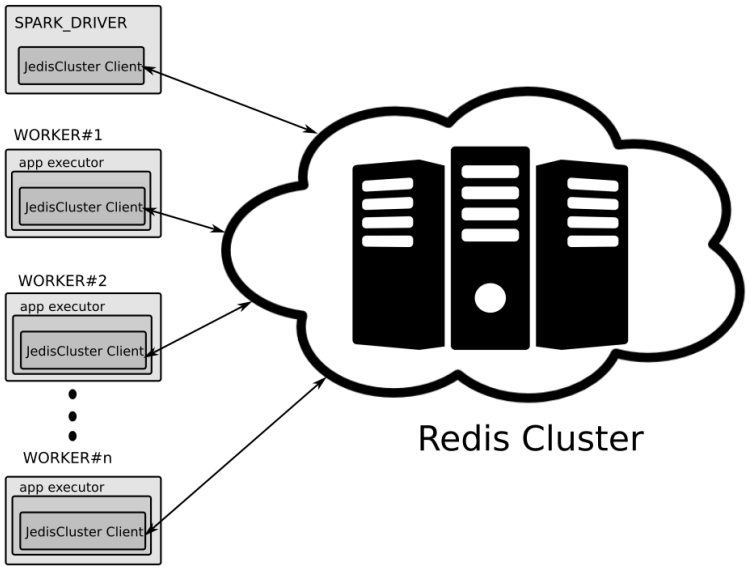
$$--conf\ spark.redis.settings.enable.min.idler = true/false$$

$$--conf\ spark.aggregated.fetch = true/false$$

$$other\ configs$$

$$script$$

Στην υλοποίηση μας ο κάθε executor όπως και ο driver δημιουργούν ένα jedis client για την επικοινωνία τους με το Redis Cluster. Η κλάση που είναι υπεύθυνη για την δημιουργία του client, είναι η RedisServerEnv η οποία έχει ως όρισμα το αντικείμενο του client.



Σχήμα 3.3: Apache Spark Redis διασύνδεση με client

Όσο αναφορά την δημιουργία του RedisServerEnv, γίνεται μέσω τις μεθόδου setClass από την πλευρά του Driver, αυτή καλείται μέσα από την κλάση StandaloneAppClient του driver και πιο συγκεκριμένα μέσα από την μέθοδο $StandaloneAppClient.onStart()$, μόνο όταν είναι αληθής η μεταβλητή περιβάλλοντος (environment variable) για την ενεργοποίηση του shuffle με το redis, $SPARK\_REDIS\_SHUFFLE\_ENABLE$ τότε δημιουργείται ένα RedisServerEnv αντικείμενο, το οποίο περιέχει το αντικείμενο του Redis Client. Από την πλευρά των worker, ο κάθε executor δημιουργεί τοπικά ένα BlockManager, τον οποίο χρησιμοποιεί και στην δημιουργία και στην προσκόμιση των Block. Κατά την δημιουργία του BlockManager λοιπόν, καλούμε την μέθοδο setClass() για να δημιουργήσουμε ένα αντικείμενο RedisServerEnv.

Η μέθοδος setClass χρησιμοποιεί όλα τα ορίσματα του πίνακα 3.3. Για την σύνδεση στο cluster χρησιμοποιεί τις μεταβλητές για την IP και το Port ενός master κόμβου του Redis και στη συνέχεια ο JedisCluster ανακαλύπτει αυτόματα όλους του υπόλοιπους masters. Η συγκεκριμένη μέθοδος χρησιμοποιεί και την κλάση GenericObjectPoolConfig του Jedis, η οποία χρησιμοποιείται για προκαθορισμένες ρυθμίσεις του client μας, έτσι

ώστε να μπορέσουμε να δημιουργήσουμε ένα pool από συνδέσεις προς τον Redis Cluster, θα αναφερθούμε σε αυτές στο κεφάλαιο των μετρήσεων.

Όπως προαναφέρθηκε, για την εγγραφή των δεδομένων στα shuffle αρχεία χρησιμοποιείται ένας shuffle writer ο οποίος αναλαμβάνει και την συμπίεση και την κρυπτογράφηση (σε περίπτωση που έχει ενεργοποιηθεί). Άρα εμείς από την πλευρά μας δεν χρειάζεται να πραγματοποιήσουμε αυτές τις λειτουργίες. Στο τέλος την εγγραφής καλεί την μέθοδο IndexShuffleBlockResolver.writeMetadataFileAndCommit() με την οποία κάνει τελική υποβολή (commit) ενός shuffle αρχείου για ένα map. Σε αυτό το σημείο, είναι που καλείται η στατική μέθοδος uploadShuffleBlock() της κλάσης RedisUpload, όπως φαίνεται στον πίνακα 3.3. Σε αυτό το σημείο ανοίγουμε το shuffle αρχείο, ως ένα InputStream, το οποίο χρησιμοποιούμε για να μεταβιβάσουμε τα δεδομένα στο Redis.

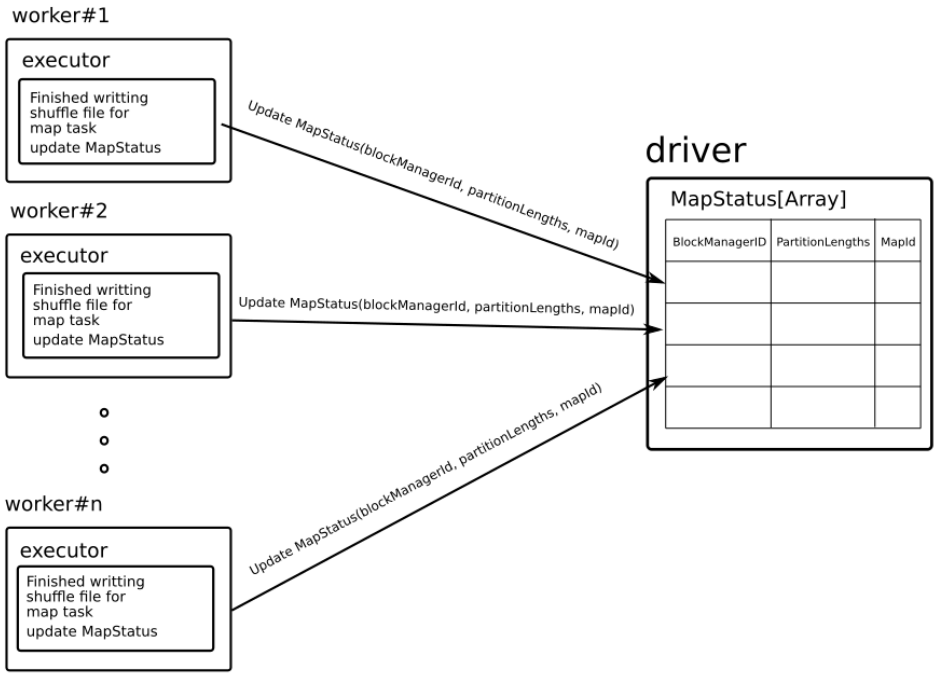
Για την μεταφόρτωση χρησιμοποιούμε την παραπάνω μέθοδο η οποία διαβάζει το Shuffle αρχείο ανά blocks (με την βοήθεια του πίνακα lengths, ο οποίος αποθηκεύει τα μεγέθη του κάθε block) και σε κομμάτια (chunks), όπου το μέγεθος του κομματιού που έχουμε επιλέξει είναι $SHUFFLE\_FILE\_BUFFER\_SIZE$, το οποίο ορίζεται στο config package. Είναι μεταβλητή περιβάλλοντος (environment variable) και η προκαθορισμένη (default) τιμή είναι 32KB. Στην συνέχεια χρησιμοποιούμε την μέθοδο uploadArray, πάλι της κλάσης RedisUpload, για την αποθήκευση των checksum του αντίστοιχού shuffle αρχείου.

Για την αποστολή των δεδομένων ως κλειδιών τιμών (key - value) στο redis χρησιμοποιούμε τις συναρτήσεις SET,APPEND του redis, όπως υποδεικνύει και το όνομα τους, η SET αναθέτει στο επιλεγμένο κλειδί την τιμή που θέτουμε, αν δεν υπάρχει ήδη το κλειδί τότε το ορίζει, ενώ η APPEND απλά προσκολλά τα δεδομένα στα ήδη υπάρχοντα δεδομένα του αναφερόμενου κλειδιού. Έτσι σε περίπτωση που το συγκεκριμένο κλειδί - block δεν έχει οριστεί, χρησιμοποιούμε την μέθοδο set και στην συνέχεια την συνάρτηση append, ώστε τα επόμενα δεδομένα που καταφθάνουν, για αυτό το κλειδί, να προσκολλάνε με την σειρά στα προηγούμενα.

Όπως αναφέραμε στο υποκεφάλαιο 3.1.1 τα blocks αποθηκεύονται στο ίδιο redis cluster node σύμφωνα με την μεθοδολογία κατακερματισμού που εφαρμόσαμε. Ο λόγος που επιλέξαμε να πραγματοποιηθεί με αυτόν τον τρόπο είναι γιατί τώρα μπορούμε αμέσως να πάρουμε το επιθυμητό block, χωρίς να χρειάζεται να προσκομίσουμε το δεικτοδοτούμενο ευρετήριο (index αρχείο) για να ενημερωθούμε για την θέση των δεδομένων μας.

Αφότου τελειώσει η μεταφόρτωση των block στο Redis, ο Shuffle Writer ενημερώνει μέσω του MapStatus [25], τον driver για την τοποθεσία του shuffle map αρχείου, δηλαδή το BlockManagerId το οποίο είναι ένα αντικείμενο που περιέχει το host IP και την πόρτα (port) του block manager που διαχειρίζεται το shuffle αρχείο, και τα μήκη σε bytes του κάθε block του συγκεκριμένου map. Το MapStatus Array αποτελεί μια δομή δεδομένων, στον driver, η οποία λειτουργεί σαν ένα κατάστιχο και φυλάει όλες τις τοποθεσίες όλων των shuffle map αρχείων, όπως και τα μήκη των επιμέρους block τους. Κατά τον τρόπο αυτό όταν μια διεργασία reduce (reduce task) πάει να διαβάσει τα block, να μπορεί να μάθει την τοποθεσία τους.

Για την ενημέρωση του MapStatus δημιουργούμε ένα ”dummy” αναγνωριστικό



Σχήμα 3.4: MapStatus ενημέρωση

BlockManagerId, για το Redis, έτσι ώστε σε περίπτωση που ενεργοποιήσουμε το redis shuffle να ενημερώσουμε το MapStatus με το dummy BlockManagerId, αντί για το BlockManager του worker / executor και χρησιμοποιείται μόνο για να μαρκάρουμε τα δεδομένα ότι βρίσκονται στο RedisCluster. Την διαχείριση της προσκόμισης block, θα την αναλύσουμε αμέσως παρακάτω.

Variable
A, blockManagerId
end Variable
if $SPARK\_REDIS\_SHUFFLE\_ENABLE$ **then**
$A \leftarrow dummyBlockManagerId$
else
$A \leftarrow blockManagerId\ of\ current\ blockManager$
end if

Στην συνέχεια ο driver διαμοιράζει σε κάθε reduce task τα απαραίτητα block, όπως και την τοποθεσία τους. Σε περίπτωση που έχει επιλεγεί το redis, τότε ως τοποθεσία θα λάβουν το dummy blockManagerId και θα πρέπει να είμαστε σε θέση να το διαχειριστούμε.

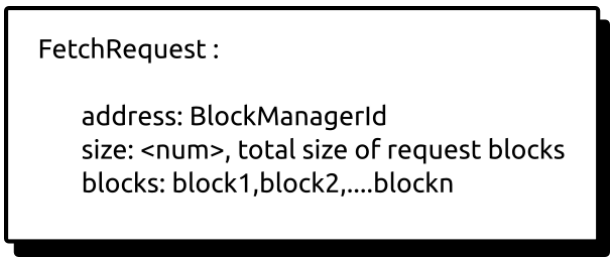
Στην συνέχεια θα αναλύσουμε την λειτουργία προσκόμισης των block μας από το Redis. Ο προκαθορισμένος τρόπος με τον οποίο ο executor του spark προσκομίζει τα

blocks από τους blockmanager βρίσκεται στην κλάση ShuffleBlockFetcherIterator, όπου η ακριβής τοποθεσία δίνεται στον πίνακα 3.1.

Τα βήματα που ακολουθεί είναι τα εξής:

1. Αρχικά προσκομίζονται από τον driver, και πιο συγκεκριμένα από το MapStatus, τα ονόματα των απαραίτητων blocks, οι τοποθεσίες τους, όπως και τα μεγέθη τους.
2. Στην συνέχεια δημιουργείται η κλάση ShuffleBlockFetcherIterator, για το reduce task, όπου γίνεται αρχικοποίηση με την βοήθεια της μεθόδου initialize()
3. Μέσα από την initialize() καλείται η μέθοδος partitionBlocksByFetchMode(), η οποία είναι υπεύθυνη για τον διαμοιρασμό και την πακετοποίηση των block σε requests προς τους BlockManagers.
4. Στην συνέχεια καλείται η μέθοδος fetchUpToMaxBytes, η οποία αναλαμβάνει να στείλει μερικά από τα request στους BlockManagers, οι οποίοι στέλνουν ασύγχρονα τα blocks στο reduce task.
5. Τέλος, καλείται η μέθοδος next(), η οποία διαβάζει τα block και επιστρέφει ένα tuple με το blockId και το inputStream, όπου blockId είναι το όνομα του block και inputStream είναι το stream των δεδομένων του block. Στο τέλος της next καλείται ξανά η μέθοδος fetchUpToMaxBytes(), έτσι ώστε να ζητήσει και άλλο request, αν αυτό είναι δυνατόν.

Στην συνέχεια θα αναλύσουμε σε περισσότερο βάθος την λειτουργία της μεθόδου partitionBlocksByFetchMode. Αρχικά, διερευνά το blockManagerId του block, αν αυτό συμπίπτει με τον τοπικό BlockManager του τοπικού worker/executor, στο οποίο βρίσκεται το Reduce Task, τότε το block μαρκάρεται για τοπική προσκόμιση. Σε διαφορετική περίπτωση, τα block συγκεντρώνονται σε requests, όπου όλα τα block ενός request ανήκουν στον ίδιο BlockManager και το μέγιστο μέγεθος του request ορίζεται από την μεταβλητή $targetRemoteRequestSize = math.max(maxBytesInFlight/5, 1L)$, όπου maxBytesInFlight είναι το μέγιστο μέγεθος από bytes τα οποία μπορούν να μεταδίδονται ταυτόχρονα. Βλέπουμε πως το targetRemoteRequestSize είναι το 1/5 του maxBytesInFlight, το οποίο σημαίνει πως ταυτόχρονα μπορούν να εκτελούνται 5 request.



Σχήμα 3.5: Spark Fetch Request

Η μέθοδος fetchUpToMaxBytes(), εξετάζει αν μπορεί να σταλθεί το request προς τον αντίστοιχο BlockManager. Με την βοήθεια των μεθόδων, isRemoteBlockFetchable

και isRemoteAddressMaxedOut. Η μέθοδος isRemoteBlockFetchable είναι υπεύθυνη για να υπολογίσει αν το εξεταζόμενο request θα ξεπεράσει τα όρια των μέγιστων μεταφερόμενων byte, όπως επίσης και των μέγιστων ταυτόχρονων request.

$$(bytesInFlight == 0 \;||$$

$$(reqsInFlight \; + \; 1 \; <= \; maxReqsInFlight$$

$$bytesInFlight \; + \; fetchReqQueue.front.size \; <= \; maxBytesInFlight))$$

Η μέθοδος isRemoteAddressMaxedOut εξετάζει εάν έχουμε υπερβεί τον μέγιστο αριθμό ταυτόχρονων request για μια συγκεκριμένη διεύθυνση.

Στην εργασία μας θα πρέπει να βρούμε ένα τρόπο να διαχειριστούμε τον dummy BlockManagerId, που έχει δημιουργηθεί για το redis, καθώς τώρα όλα τα block μας είναι αντιστοιχισμένα σε αυτόν. Για τον σκοπό αυτόν έχει δημιουργηθεί προσαρμοσμένος τρόπος έτσι ώστε να μπορέσουμε να διαχειριστούμε τον dummy Block Manager.

Διαχειριζόμαστε το διάβασμα των block με την δημιουργία ενός τρόπου προσκόμισης περισσότερων ή λιγότερων από 5 ταυτόχρονων request, καθώς αυτό είναι το όριο που θέτει το spark από μόνο του. Αυτό επιτυγχάνεται με 2 βασικά στοιχεία:

1. Την μεγέθυνση ή συρρίκνωση σε μέγεθος των request, για την ταυτόχρονή προσκόμιση περισσότερων ή λιγότερων σε πλήθος request. Αυτό γίνεται στην μέθοδο collectFetchRequests, όπου πραγματοποιείται το άθροισμα των block σε request. Το άθροισμα το block γίνεται με βάση τα μεγέθη τους και πιο συγκεκριμένα ένα block έχει μέγιστο κατώφλι από bytes, maxBytesInFlight / 5, που σημαίνει ότι ταυτόχρονα μπορεί να έχει το πολύ 5 requests προς προσκόμιση αν υποθέσουμε ότι τα request έχουν το μέγιστο μέγεθος. Στην περίπτωση μας, επειδή θέλουμε να μην έχουμε σταθερό αριθμό από request προς προσκόμιση ταυτόχρονα, μικραίνουμε ή μεγαλώνουμε το μέγεθος του request σε maxBytesInFlight / SPARK_REDIS_MAX_REQS_PER_EXECUTOR, όπου ο παρανομαστής είναι ο αριθμός των συνδέσεων προς κάθε κόμβο του redis cluster και το ορίζεται σύμφωνα με τον πίνακα 3.3.

2. Δημιουργούμε την μέθοδο fetchUpToMaxBytesRedis(), η οποία είναι υπεύθυνη για την δουλειά του fetchUpToMaxBytes() για τα request του redis. Η μέθοδος fetchUpToMaxBytesRedis ελέγχει, με βάση την συνθήκη (bytesInFlight == 0 || (reqsInFlight + 1 <= conf.get(SPARK_REDIS_MAX_REQS_PER_EXECUTOR) && bytesInFlight + fetchReqQueue.front.size <= maxBytesInFlight)), αν το request είναι προσκομήσιμο. Η κύρια διαφορά με την συνθήκη του fetchUpToMaxBytes() είναι πως εδώ ο μέγιστος αριθμός request μπορεί να είναι ίσος με την μεταβλητή SPARK_REDIS_MAX_REQS_PER_EXECUTOR (το οποίο όπως αναφέραμε είναι ο μέγιστος αριθμός συνδέσεων προς κάθε redis κόμβο). Αν το request περάσει το test, τότε καλείτε η μέθοδος fetchFromRedis ή η μέθοδος fetchFromRedisPipeline για να γίνει ασύγχρονη προσκόμιση των block.

Η μέθοδος fetchFromRedis αρχικά δημιουργεί ένα thread, έτσι ώστε να προσκομίσει τα block ώστε να μην καθυστερεί το υπόλοιπο πρόγραμμα. Τα request που έρχονται δεν ξεπερνάνε το μέγιστο μέγεθος maxReqSizeShuffleToMem, το οποίο ορίζει το μέγιστο μέγεθος ενός request που χωράει στην μνήμη. Στους ελέγχους μας έχουμε

χρησιμοποιήσει block μεγέθους 3KB και 30KB, άρα το μέγεθος του request που ορίστηκε ως maxBytesInFlight / SPARK_REDIS_MAX_REQS_PER_EXECUTOR δεν θα ξεπεράσει το κατώφλι του maxReqSizeShuffleToMem. Από προεπιλογή το spark ορίζει maxBytesInFlight 48MB και το μέγεθος maxReqSizeShuffleToMem σε 200MB. Στην προκειμένη περίπτωση, όπου τα πακέτα είναι μικρά, δεν θα ξεπεραστεί ποτέ. Άρα βάζουμε τα πακέτα κατευθείαν στην μνήμη. Οπότε δημιουργούμε ένα for-loop και προσκομίζουμε ένα-ένα τα block, στην συνέχεια τυλίγουμε (wrap) τα δεδομένα γύρω από ένα Netty ByteBuf (καθώς χρησιμοποιούμε την κλάση NettyManagedBuffer που χρησιμοποιεί και το spark) και στην συνέχεια ενημερώνουμε της ουρά results, η οποία κρατάει όλα τα προσκομισμένα block που δεν έχουν επεξεργαστεί ακόμα.

Αν και από τους ελέγχους μας τα πακέτα, λόγω του μικρού μεγέθους, δεν ξεπερνάνε το κατώφλι του maxReqSizeShuffleToMem, παρ' όλα αυτά έχει υλοποιηθεί και η περίπτωση στην οποία το request ξεπερνάει το μέγεθος του κατωφλίου. Αν και σε αυτό το σημείο αυτό πρέπει να σημειωθεί πως αυτή η περίπτωση δεν έχει ελεγχθεί πλήρως, καθώς όπως προαναφέραμε δεν ξεπερνάμε αυτό το κατώφλι. Στην περίπτωση που το κατώφλι του maxReqSizeShuffleToMem ξεπεραστεί προσκομίζουμε τα πακέτα και τα αποθηκεύουμε σε κάποιο προσωρινό αρχείο. Αν το πακέτο είναι μικρότερο ή ίσο του μεγέθους με το κατώφλι maxReqSizeShuffleToMem τότε απλά το προσκομίζουμε και το προσθέτουμε σε ένα προσωρινό αρχείο, σε περίπτωση που είναι μεγαλύτερο, αναζητούμε το μέγεθος από το jedis (καθώς το μέγεθος που μας παρέχεται από τον driver είναι προσεγγιστικό και όχι ακριβές, καθώς κατά την μεταφορά του από τον driver συμπιέζεται). Στη συνέχεια το προσκομίζουμε σε κομμάτια ίσα με το μέγεθος maxReqSizeShuffleToMem τα οποία αποθηκεύουμε σε ένα προσωρινό αρχείο.

Η μέθοδος fetchFromRedisPipeline αρχικά δημιουργεί ένα thread, έτσι ώστε να μην απασχολεί το υπόλοιπο πρόγραμμα, **υποθέτουμε επίσης ότι όλα τα request πάλι χωράνε στην μνήμη, όπως παραπάνω. Επίσης, όπως αναφέραμε και στην αρχή, έχουμε κάνει την υπόθεση πως οι κόμβοι του redis δεν χάνονται ούτε προστίθενται νέοι κόμβοι, πράγμα που θα είχε σαν αποτέλεσμα την αλλαγή του διαμοιρασμού των θέσεων (slot) στο cluster**. Η διαφορά, με την προηγούμενη περίπτωση, είναι πως εδώ χρησιμοποιούμε την μέθοδο του pipeline για την προσκόμιση των block. Όπως αναφέραμε το redis αποτελείται από 16384 θέσεις (slots). Αυτές οι θέσεις διαμοιράζονται σε κάθε κόμβο του redis. Αρχικά, με την μέθοδο του jedis clusterSlots ενημερωνόμαστε για τα slot του κάθε κόμβου. Έπειτα, σειριακά ανοίγουμε κατευθείαν σύνδεση προς κάθε κόμβο. Ξεχωριστά και μέσω pipeline στέλνουμε εντολές get για όλα τα block τα οποία έχουν αριθμό slot στο διάστημα του κόμβου. Το pipeline μας επιτρέπει να στείλουμε πολλαπλά αιτήματα (request, δεν είναι τα ίδια με τα request των block, εδώ αναφερόμαστε στα αιτήματα προς τον redis cluster), ακόμα και αν οι απαντήσεις δεν έχουν φτάσει[26]. Έτσι, στέλνουμε όλα τα αιτήματα get για τα block μας και στο τέλος περιμένουμε για την απάντηση από όλα με την μέθοδο sync(). Στην συνέχεια ενημερώνουμε το result queue με τα block και προχωράμε στον επόμενο κόμβο.

Algorithm 1 fetchFromRedis Algorithm

```
procedure FETCHFROMMONGO(dbFetchRequest)
    start thread
    if dbFetchRequestmaxReqSizeShuffleToMem then
        for block in dbFetchRequest.blocks do
            if block.size < maxReqSizeShuffleToMem then
                buf ← JedisCluster.get(block.name)
                save buf to file
                update result queue with new block
            else
                realBlockLength ← JedisCluster.length(block.name)
                bytes_fetched ← 0
                while(bytes_fetched != realBlockLength) do
                    chunk ← JedisCluster.getRange(block.name, bytes_fetched, bytes_fetched+
                            maxReqSizeShuffleToMem)
                    bytes_fetched ← bytes_fetched + chunk.length
                    save chunk to file
                end while
                update result queue with new block
            end if
        end for
    else
        for block in dbFetchRequest.blocks do
            buf ← JedisCluster.get(block.name)
            update result queue with new block
        end for
    end if
end procedure
```

Algorithm 2 fetchFromRedisPipeline Algorithm

```
procedure FETCHFROMREDISPIPELINE(dbFetchRequest)
    start thread
    nodeSlots ← Jedis.clusterSlots
    for node in jedis nodes do
        respMap < Block, resp >← {}
        for Block in request.blocks do
            blockSlot ← jedis.slotKey(Block)
            if blockSlot>=nodeSlots(0) and blockSlot<=nodeSlots(1) then
                respMap(Block) ← jedisPipeline.get(Block)
            end if
        end for
        jedisPipeline.sync()
        for record in respMap do
            put result in result Queue
        end for
    end for
end procedure
```

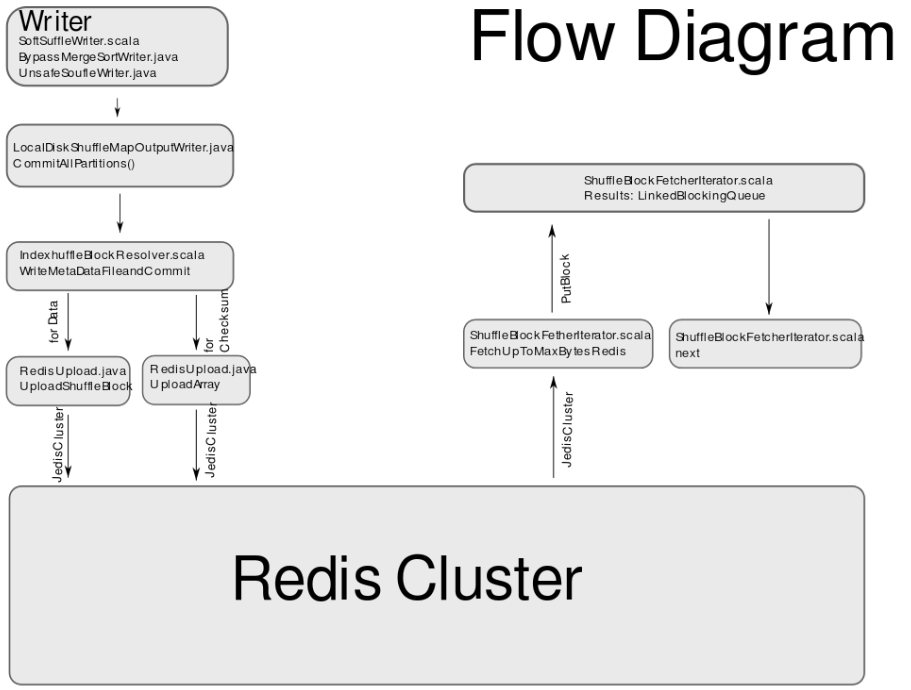
Algorithm 3 fetchUpToMaxBytesRedis Algorithm

```
dbFetchRequests, queue of requests
procedure FETCHUPTOMAXBYTESREDIS
    while ISREMOTEBLOCKFETCHABLE(dbFetchRequests) do
        request ← DEQUEUE(dbFetchRequests)              ▷ Dequeue request of head
        FETCHFROMREDIS(request) / FETCHFROMREDISPIPELINE(request)
    end while
    procedure ISREMOTEBLOCKFETCHABLE(dbFetchRequests)
        return dbFetchReuests.notEmpty && (dbReqsInFlight + 1 <=
        SPARK_REDIS_MAX_REQS_PER_EXECUTOR
        bytesInFlight + fetchReqQueue.front.size <= maxBytesInFlight)
    end procedure
end procedure
```

Όταν τελειώσει η εφαρμογή του redis, ο driver αναλαμβάνει να διαγράψει όλα τα blocks από την μνήμη του redis cluster. Αυτό το πετυχαίνει με την βοήθεια της κλάσης RedisRemoveBlocks και πιο συγκεκριμένα της μεθόδου RemovePattern, η οποία αναλαμβάνει να διαγράψει όλα τα block που αρχίζουν με το πρόθεμα { + appId + /. καθώς όπως έχουμε αναφέρει παραπάνω και πιο συγκεκριμένα στο υποκεφάλαιο 3.1.1. Όλα τα blocks της ίδιας εφαρμογής αρχίζουν από το ίδιο πρόθεμα, αυτό το πετυχαίνουμε βάσει του αλγορίθμου [27] όπου, όπως και στο διάβασμα του pipeline, προπελαύνουμε κάθε κόμβο και αναζητούμε όλα τα ζευγάρια κλειδιού τιμής που έχουν στο κλειδί τους το παραπάνω αναφερόμενο πρόθεμα, με την βοήθεια της συνάρτησης scan του redis η οποία μας βοηθάει στην προσπέλαση των κλειδιών. Έπειτα, όσα κλειδιά έχουν το αναφερόμενο πρόθεμα τα βάζουμε σε μία λίστα και έπειτα με την βοήθεια του pipeline τα διαγράφουμε, στην συνέχεια συνεχίζουμε στον επόμενο κόμβο μέχρις ότου τελειώσουμε.

Algorithm 4 removePattern Algorithm

```
dbFetchRequests, queue of requests
procedure REMOVEPATTERN
    pattern ← {appId/*
    nodeSlots ← Jedis.clusterSlots
    for node in jedis nodes do
        cursor ← jedis.scan(pattern)
        keyList ← {}
        while cursor not done do
            add to keyList all keys with pattern
        end while
        for key in keyList do
            jedisPipeline.del(key)
        end for
        jedisPipeline.sync()
    end for
end procedure
```



Σχήμα 3.6: Διάγραμμα ροής κώδικα

Στο διάγραμμα 3.6 παρουσιάζεται αναλυτικότερα η ροή των δεδομένων του shuffle για το redis, με βάση την προαναφερόμενες σ' αυτό το κεφάλαιο κλάσεις.

1. Αρχικά ο write ξεκινάει να γράφει τα δεδομένα.
2. Μετά την ολοκλήρωση, καλείται η μέθοδος commitAllPartition που βρίσκεται στον IndexShuffleBlockResolver, όπου και γίνεται το upload με την βοήθεια του Jedis στον Redis Cluster
3. Κατά την ανάγνωση καλούμε την ειδικά διαμορφωμένη fetchUpToMaxBytesRedis, η οποία πάλι μέσω του JedisCluster προσκομίζει τα block, τα οποία τα τοποθετεί σε μια ουρά και έπειτα η μέθοδος next αναλαμβάνει την παραχώρηση τους προς επεξεργασία.

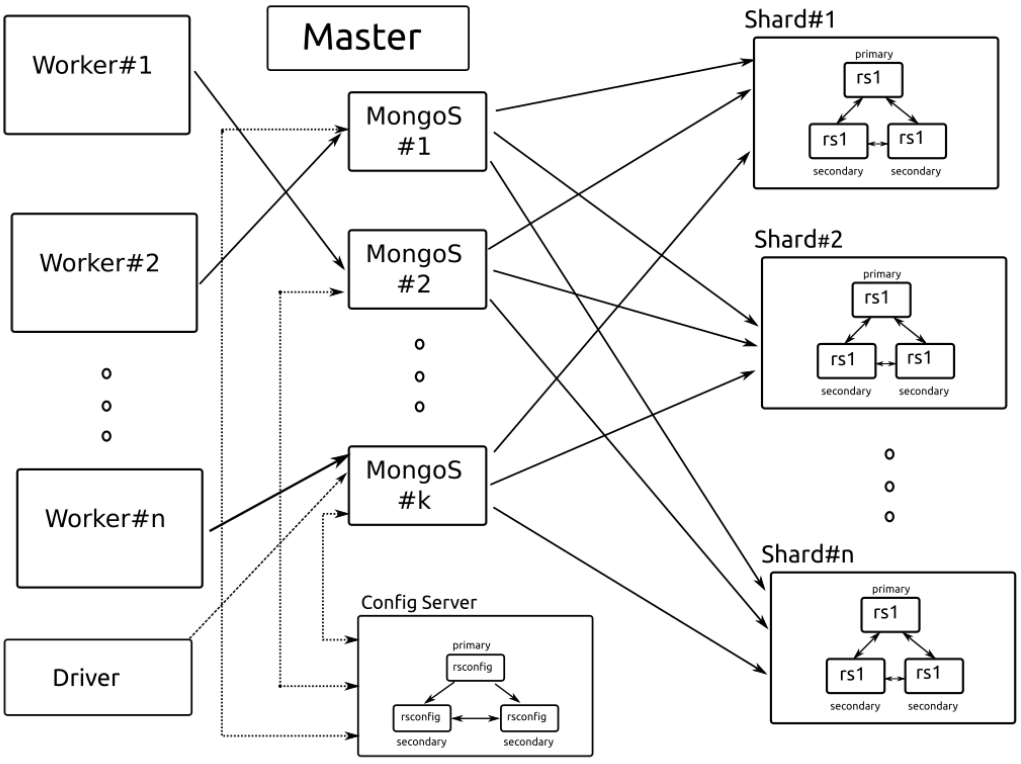
3.2 Επέκταση του shuffle service με mongodb

Στο παρόν υποκεφάλαιο θα παρουσιάσουμε την επέκταση του shuffle service με την βοήθεια της μη σχεσιακής βάσης δεδομένων mongodb, πράγμα που σημαίνει πως τα ενδιάμεσα δεδομένα αποθηκεύονται, ως ένα σύνολο από επιμέρους κομμάτια (chunks), πάνω στην βάση. Όπως και στο υποκεφάλαιο του Redis, αξίζει να σημειωθεί πως η παρούσα υλοποίηση με το mongodb αποτελεί ένα proof-of-concept και έχει εξεταστεί με το stress test workload που παρουσιάζεται στο υποκεφάλαιο 4.3. Επίσης πρέπει να σημειωθεί πως υποθέτουμε ότι κατά την διάρκεια της εκτέλεσης δεν χάνονται κόμβοι και κατά συνέπεια δεν έχουμε απώλεια ενδιάμεσων δεδομένων κατά

την διάρκεια του shuffle από την πλευρά της βάσης. Αυτό θα είχε ως αποτέλεσμα να έπρεπε να διαχειριστούμε fetch fail exception, τα οποία παράγει το spark σε περίπτωση που δεν μπορεί να διαβάσει τα δεδομένα κάποιου block.

3.2.1 Λειτουργία του shuffle με το mongodb

Στην εργασία αυτή το mongodb έχει παραταχθεί σε μορφή distributed sharded cluster, που σημαίνει ότι υπάρχει η επιλογή να διαμοιράσουμε τα δεδομένα μιας συλλογής (collection) στα διαφορετικά Shard του cluster μας, επί τη βάση κάποιου κλειδιού διαμοιρασμού. Αυτό το κλειδί διαμοιρασμού μπορεί να είναι είτε ένα σκέτο πεδίο (field) της συλλογής (collection), το οποίο είναι ευρετηριασμένο (indexed), είτε πολλαπλά πεδία που καλύπτονται από ένα συνθετικό ευρετήριο (coumpound index)[28]. Τα ευρετήρια υποστηρίζουν την αποτελεσματική εκτέλεση ερωτημάτων (Queries). Χωρίς αυτά το mongodb θα έπρεπε να συλλέξει όλα τα έγγραφα(document) μιας συλλογής (collection) για την απάντηση του ερωτήματος [29].



Σχήμα 3.7: Spark - Mongodb Block Διάγραμμα

Στην εργασία οι Worker και ο Driver όπως φαίνεται και στο σχήμα συνδέονται μέσω κάποιου διακομιστή mongos στο Mongodb Sharded Cluster, έτσι ώστε να επεκτείνουν το shuffle3.7. Στην συνέχεια, για την μεταφόρτωση των αρχείων του shuffle στο mongodb, δημιουργήθηκαν 2 συλλογές, όπου η μια αποθηκεύει τα blocks του shuffle και η δεύτερη αποθηκεύει τα checksums του κάθε shuffle αρχείου.

Όσο αναφορά το μέγεθος των αρχείων θα πρέπει να είμαστε προσεκτικοί, καθώς κάθε εγγραφή του mongoDB είναι τύπου BSON. Το BSON έχει μέγιστο μέγεθος

εγγραφής 16MB και έτσι για την αποθήκευση μεγάλων αρχείων θα πρέπει να διασπαστούν σε κομμάτια (chunks). Η συλλογή για τα block του shuffle block προσφέρει την δυνατότητα διαχωρισμού σε chunk, των οποίων το μέγεθος το ορίζουμε εμείς, μέσω του ορίσματος spark.mongodb.chunk.size.

Συλλογές που δημιουργούνται:

1. Η συλλογή shuffleBlock που αποθηκεύει τα block του shuffle σε κομμάτια (chunks)
2. Η συλλογή shuffleMetadata που αποηθεκεύει τα checksum των shuffle αρχείων

Επεξήγηση πεδίων της συλλογής shuffleBlocks :

1. Το πεδίο _id αναφέρεται στο μοναδικό αναγνωριστικό του εγγράφου document.
2. Το πεδίο length περιέχει τον μέγεθος του παρόντος chunk.
3. Το πεδίο n προσδιορίζει την σειρά των κομματιών του block.
4. Το πεδίο shuffleName περιέχει το string που αναφέρεται στο shuffleId και mapId του συγκεκριμένου block π.χ. shuffle_0_0, shuffleId = 0, mapId = 0
5. Το πεδίο metadataId περιέχει το μοναδικό αναγνωριστικό του document με τα metadata του συγκεκριμένου shuffle αρχείου (_id)
6. Το πεδίο reduce αναφέρεται στο reduceId του συγκεκριμένου block.
7. Το πεδίο data περιέχει τα δεδομένα του chunk του block του shuffle
8. Το πεδίο next μας ενημερώνει αν υπάρχει επόμενο chunk για το συγκεκριμένο block.

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "n" : <num>,
  "shuffleName" : <string>,
  "metadataId" : <ObjectId>,
  "reduce" : <num>,
  "data" : <Binary>,
  "next" : <Boolean>
}
```

Επεξήγηση πεδίων της συλλογής shuffleMetadata:

1. Το πεδίο _id αναφέρεται στο μοναδικό αναγνωριστικό του εγγράφου document.
2. Το πεδίο shuffleName περιέχει το string που αναφέρεται στο shuffleId και mapId των παρόντων checksum π.χ. shuffle_0_0, shuffleId = 0, mapId = 0.

3. Το πεδίο checksums αναφέρεται στον πίνακα με τα checksum του συγκεκριμένου shuffle αρχείου.
4. Το πεδίο Date αντιπροσωπεύει την ημερομηνία δημιουργίας του Document
5. Το πεδίο checksumKind αναφέρεται στον αλγόριθμο που χρησιμοποιήθηκε για την εξαγωγή των checksum.

```
{
  "_id" : <ObjectId>,
  "shuffleName" : <String>,
  "checksums" : <num[]>,
  "date" : <Date>,
  "checksumKind" : <string>,
}
```

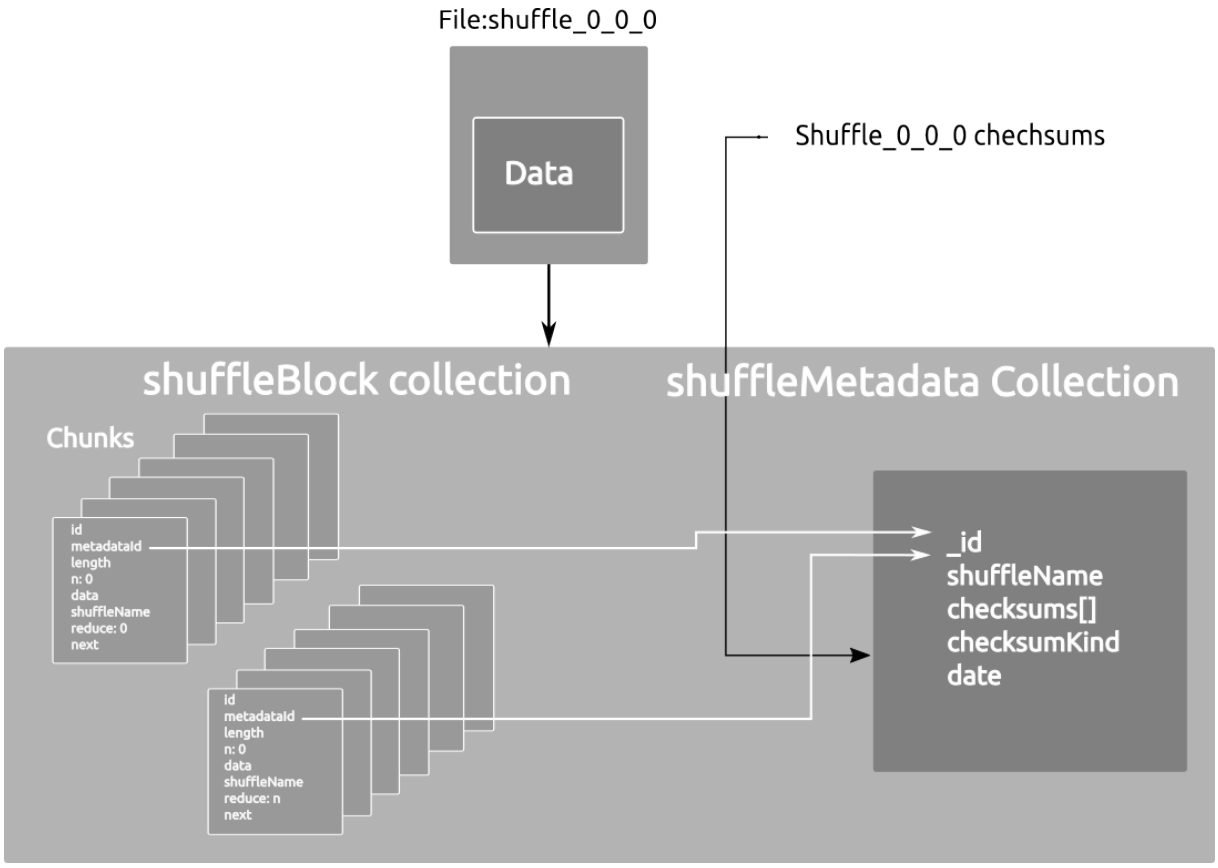
Ακολουθήσαμε μια αρχιτεκτονική στην δημιουργία των συλλογών μας παρόμοια με αυτή του gridFS[30], όπου δημιουργούνται 2 συλλογές μια για τα μεταδεδομένα του shuffle αρχείου και μια για τα κομμάτια (chunks) των block του shuffle αρχείου, με την διαφορά ότι εμείς δεν αποθηκεύουμε ολόκληρο το αρχείο σε chunks αλλά αποθηκεύουμε κάθε block ως ξεχωριστά κομμάτια(chunk) αν χρειαστεί.

Άρα με την μεταφόρτωση ενός αρχείου shuffle από την διεργασία map δημιουργείται ένα έγγραφο (document) στην συλλογή shuffleMetadata όπου και αποθηκεύονται μεταδεδομένα και τα checksums του shuffle αρχείου. Δημιουργούνται εγγραφές στην συλλογή shuffleBlock, για την αποθήκευση των block του εκάστοτε αρχείου. Στην καλύτερη περίπτωση δημιουργούνται εγγραφές ίσες με τον αριθμό των partition του shuffle, αν κάθε partition δεν ξεπερνάει το προκαθορισμένο μέγεθος (chunk size). Σε διαφορετική περίπτωση, αν το μέγεθος ενός partition ξεπερνάει το μέγεθος του κομματιού (chunk), τότε δημιουργούνται $\lceil \frac{partitionLength}{chunkSize} \rceil$ όπου όλα τα chunk, εκτός του τελευταίου, θα έχουν το προκαθορισμένο μέγεθος (chunkSize), εκτός του τελευταίου το οποίο θα έχει $partitionLength \bmod chunkSize$.

Στην συνέχεια, δημιουργούμε και ένα ευρετήριο (index) για την συλλογή shuffleBlock το οποίο περιέχει ως 3 βασικά πεδία το metadataId, το reduce και το n, και ένα ευρετήριο για την συλλογή shuffleMetadata στο πεδίο shuffleName.

```
runCommand {
   createIndex: "<database>.shuffleBlock",
   fields: {{metadataId:1, reduce:1, n:1}},
}

runCommand {
   createIndex: "<database>.shuffleMetadata",
   fields: {{shuffleName:1}},
}
```



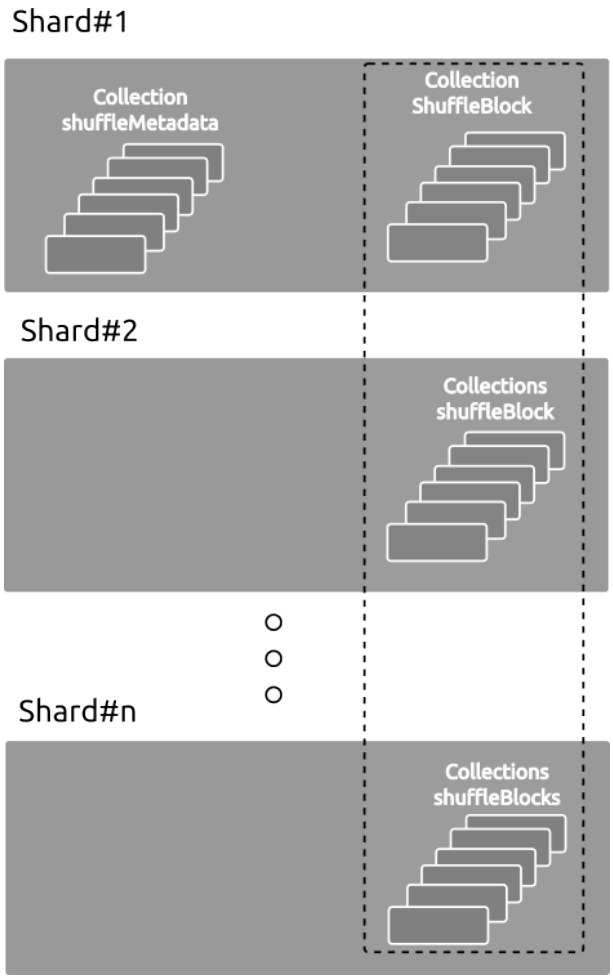
Σχήμα 3.8: Mongo Collection - Αρχείο Shuffle Διάγραμμα

Για τον διαμοιρασμό λοιπόν των αρχείων σε διαφορετικά Shard του mongodb cluster, πρέπει να ορίσουμε μια συλλογή ως sharded, έτσι ώστε το mongodb να διαμοιράσει τα δεδομένα στα επιμέρους shards. Όπως προαναφέραμε δημιουργούμε 2 κλάσεις την shuffleMetadata και shuffleBlock, επιλέγουμε να μετατρέψουμε την συλλογή shuffleBlock σε sharded συλλογή με την παρακάτω εντολή:

```
1 runCommand {
2   shardCollection: "<database>.shuffleBlock",
3   key: { metadataId: <1|"hashed">},
4 }
```

Το sharding γίνεται βάση του πεδίου metadataId. Το metadataId, όπως προαναφέρθηκε, είναι το μοναδικό αναγνωριστικό ολόκληρου του shuffle αρχείου (δηλαδή είναι το ObjectId του shuffle αρχείου της συλλογής shuffleMetadata), οπότε όλα τα chunk όλων των partition που ανήκουν στο ίδιο shuffle αρχείο θα έχουν το ίδιο metadataId, το οποίο αποτελεί το κλειδί κατακερματισμού (shard key) με πολιτική hashing.

Σε περίπτωση που υπάρξει ανισορροπία στην κατανομή των δεδομένων μιας sharded συλλογής, το mongodb διαθέτει τον balancer, ο οποίος είναι μια λειτουργία που τρέχει στο παρασκήνιο, αναλαμβάνει να ισοσταθμίσει την ανισορροπία μεταξύ των shard, αν υπάρχει. Ένα sharded key αποτελείται από chunks(τα οποία είναι διαφορετικά από



Σχήμα 3.9: collections στο sharded cluster

τα chunk της συλλογής shuffleBlock) και αποτελείται από ένα εύρος κλειδιών που αναλαμβάνουν να φυλάξουν το κάθε shard. Σε περίπτωση που σε κάποιο το μέγεθος των δεδομένων ενός chunk μεγαλώσει πολύ, ο balancer αναλαμβάνει να διασπάσει το chunk και να "μεταναστεύσει" τα δεδομένα τους.

Εν κατακλείδι, θα αναφερθούμε συνοπτικά στην διαδικασία που ακολουθούμε για την χρήση του mongodb ως ενδιάμεσου εργαλείου αποθήκευσης shuffle.

1. Αρχικά ο spark driver δημιουργεί μια βάση δεδομένων, με όνομα το id της εφαρμογής του spark.
2. Ο spark driver δημιουργεί την βάση με όνομα appId όπου appId είναι το όνομα της εφαρμογής του spark και μέσα σε αυτή δημιουργεί τις βάσεις shuffleBlock και shuffleMetadata.
3. Ο spark driver δίνει εντολή στο mongodb να δημιουργήσει ευρετήρια (indexes) πάνω στις συλλογές και πιο συγκεκριμένα στην συλλογή shuffleBlock, δημιουργείται το ευρετήριο με τα πεδία (metadata:1, reduce:1, n:1).
4. Ο spark driver δίνει εντολή στο mongodb να γίνει shard η συλλογή shuffleBlock στο πεδίο metdataId.

5. Οι worker χρησιμοποιούν τον driver του mongodb για μεταφόρτωση και προσκόμιση των shuffle blocks.

6. Μετά το πέρας της spark εφαρμογής ο driver διαγράφει την βάση.

3.2.2 Υλοποίηση του shuffle με το mongodb στο Apache Spark

Όπως και στο υποκεφάλαιο 3.1.3 η επέκταση του shuffle service του spark με το mongodb έχει ως αποτέλεσμα την τροποποίηση-διαμόρφωση των αναφερόμενων πηγαίων αρχείων κλάσεων 3.1. Στην συνέχεια παραθέτουμε τα αρχεία πηγαίου κώδικα που δημιουργήθηκαν για την επέκταση του shuffle με το mongodb:

Κλάση	Package path	Relative path from
MongodbServerEnv	org.apache.spark .MongodbServerEnv	core/src/main/scala/org/ apache/spark/ MongodbServerEnv.scala
MongodbUpload	org.apache.spark.shuffle .mongodb.MongodbUpload	core/src/main/scala/org/ apache/spark/shuffle/ mongodb/ MongodbUpload.java

Πίνακας 3.4: Πίνακας χρήσιμων κλάσεων Mongodb Apache Spark

Η κλάση MongodbServerEnv διαθέτει τον mongodb java client, με τον οποίο επικοινωνούμε με το Mongodb Sharded Cluster, παράλληλα διαθέτει και δύο αντικείμενα MongoCollection για τις συλλογές που έχουμε δημιουργήσει (shuffleMetadata και shuffleBlock).

Σε αυτό το σημείο επεκτείνουμε, περαιτέρω, το package config, προσθέτοντας και μερικές custom μεταβλητές, έτσι ώστε να μπορέσουμε να ενημερώσουμε το spark, από το command line του spark-submit, ώστε να χρησιμοποιήσει το Mongodb ως κύριο μηχανισμό για το shuffle, αλλά και να ρυθμίσουμε το client στα μέτρα μας. Παρακάτω παρουσιάζουμε τις environment μεταβλητές που προστέθηκαν στο config package, σε μορφή πίνακα, όπου η πρώτη στήλη αντιπροσωπεύει το όνομα της μεταβλητής εσωτερικά, η δεύτερη στήλη αντιπροσωπεύει το όνομα για το εξωτερικό configuration είτε από αρχείο, είτε από την γραμμή εντολών, η τρίτη στήλη αντιπροσωπεύει την default τιμή της μεταβλητής και η τέταρτη στήλη περιγράφει συνοπτικά την χρήση την μεταβλητής.

Environment Μεταβλητή	Command line όνομα	default τιμή	Χρήση
SPARK_MONGODB_SHUFFLE_ENABLE	spark.mongodb .shuffle.enable	False	Ενεργοποίηση του shuffle με mongodb
SPARK_MONGODB_SHUFFLE_CONNECTION_STRING	spark.mongodb.shuffle .connection.string	no default	Connection String από πολλά mongo services θα επιλεγεί ανάλογα με το Id του executor
SPARK_MONGODB_MAX_REQS_PER_EXECUTOR	spark.mongodb.max .reqs .per.executor	10	Αριθμός ταυτόχρονων συνδέσεων με το mongo service
SPARK_MONGODB_CHUNK_SIZE	spark.mongodb.chunk .size	261120	Μέγεθος chunk του block σε bytes
SPARK_MONGODB_MIN_CONNECTIONS	spark.mongodb.min .connections	false	false == 1 min idle connection, true == min = max idle connections
SPARK_BATCH_ENTRY	spark.batch .entry	1000000	αριθμός buffered bytes προς μεταφόρτωση
SPARK_MONGODB_MAX_WAIT_TIME	spark.mongodb .max.wait.time	120000	αριθμός σε ms αναμονής της σύνδεσης

Πίνακας 3.5: Πίνακας environment μεταβλητών Mongodb

Παράδειγμα εκτέλεσης spark - submit για ενεργοποίηση του Redis

$$spark - submit\ master://master\_ip:master\_port$$

$$--conf\ spark.mongodb.shuffle.enable = true$$

$$--conf\ spark.mongodb.shuffle.connection.string =$$

$$"mongodb://mongos1IP://Port, mongodb://mongos2IP:Port..."$$

$$--conf\ spark.mongodb.max.reqs.per.executor = ConnectionNum$$

$$--conf\ spark.mongodb.chunk.size = 261120$$

$$--conf\ spark.batch.entry = 1000000$$

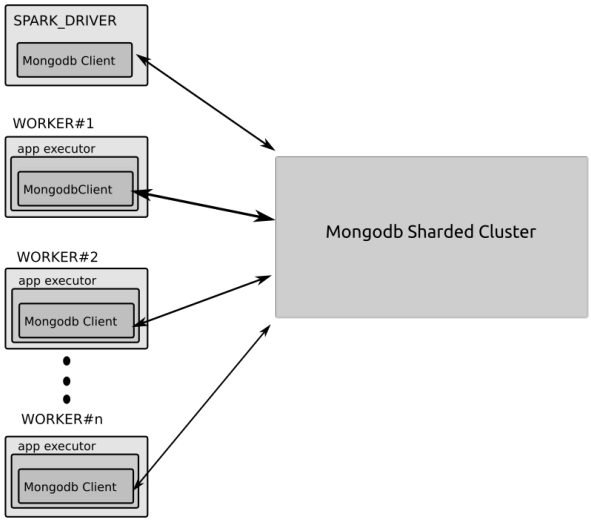
$$--conf\ spark.mongodb.max.wait.time = 120000$$

$$other\ configs$$

$$script$$

Αντιστοίχως όπως και στο redis ο κάθε executor και ο driver δημιουργούν ένα client για την επικοινωνία του με το Mongodb Cluster. Και όπως αναφέραμε πριν, η κλάση που δημιουργεί τον client στο spark είναι η MongodbServerEnv.

Όσο αναφορά την δημιουργία του MongodbServerEnv, αυτό γίνεται μέσω της μεθόδου setClass(), αντίστοιχα όπως και στην περίπτωση του redis. Ο driver καλεί αυτήν την μέθοδο, όταν λάβει απάντηση από τον master πως η εφαρμογή του έχει καταχωρηθεί προς εκτέλεση (στο σημείο $StandAloneAppClient.Receive() -> RegisteredApplication$, μόνο όταν η μεταβλητή περιβάλλοντος (environment variable) για την ενεργοποίηση του shuffle με mongodb $SPARK\_MONGODB\_SHUFFLE\_ENABLE$ είναι αληθής. Σε αυτό το σημείο ο driver αναλαμβάνει την δημιουργία μιας νέας βάσης δεδομένων, η οποία θα φιλοξενήσει τις 2 συλλογές, οι οποίες έχουν όνομα shuffleBlock και shuffleChecksum. Το όνομα της βάσης αυτής είναι appId, δηλαδή έχει το όνομα της παρούσας εφαρμογής. Επίσης, σε αυτό το σημείο, ο driver αφού δημιουργήσει τις συλλογές, δημιουργεί και τα ευρετήρια (index) τους, όπως ορίστηκαν παραπάνω, όπως επίσης στέλνει και εντολή στο mongodb cluster έτσι ώστε η συλλογή shuffleBlock να γίνει sharded. Από την μεριά του worker, όπως προαναφέραμε, ο κάθε executor δημιουργεί ένα MongodbServerEnv αντικείμενο που περιέχει τον client του mongodb. Αυτό πραγματοποιείται στον τοπικό BlockManager που δημιουργεί στην εκκίνηση ο executor.



Σχήμα 3.10: Apache Spark Mongodb διασύνδεση με client

Η μέθοδος MongodbServerEnv.setClass χρησιμοποιεί όλα τα ορίσματα του πίνακα 3.5. Για την σύνδεση στο cluster διαλέγει ένα τυχαίο connection sting ενός mongoS (οι διακομιστές με τους οποίους επικοινωνεί ο client). Παράλληλα χρησιμοποιούμε και

την μεταβλητή SPARK_MONGODB_MAX_REQS_PER_EXECUTOR για να ορίσει έναν αριθμό ταυτόχρονων συνδέσεων.

Παράλληλα επεκτείνουμε και την μέθοδο writeMetadataFileAndCommit() της μεθόδου IndexShuffleBlockResolver, την οποία όπως αναφέραμε, χρησιμοποιεί ο shuffle writer για το τελικό commit του shuffle αρχείου. Αν το mongodb shuffle είναι ενεργοποιημένο, καλείται η μέθοδος uploadStaticBlock() της κλάσης MongodbUpload για να μεταφορτώσει το shuffle αρχείο στο sharded mongodb cluster και πιο συγκεκριμένα στην sharded συλλογή shuffleBlock. Η μέθοδος αυτή δέχεται ως όρισμα το αρχείο shuffle σε μορφή stream, όπως επίσης και τα μεγέθη του κάθε shuffle block του shuffle αρχείου. Οπότε αρχικά δημιουργούμε ένα document για την συλλογή shuffleMetadata, που περιέχει τη ημερομηνία δημιουργίας του document, όπως επίσης το shuffleName, που περιέχει το shuffleId και το mapId και τα checksum, σε περίπτωση που είναι ενεργοποιημένη η λειτουργία του checksum. Στην συνέχεια διαβάζουμε τα δεδομένα από κάθε block ξεχωριστά, από το stream, σε κομμάτια (chunks), τα οποία μετατρέπουμε σε μορφή εγγράφου (document) προς αποθήκευση στη συλλογή shuffleBlock. Προσθέτουμε αυτά τα document σε μια λίστα, μέχρι ότου να φτάσουμε το όριο των μέγιστων byte που έχουμε θέσει (ορίζεται από το όρισμα spark.batch.entry και η προκαθορισμένη τιμή είναι 1000000 bytes), έτσι ώστε να κάνουμε ένα μαζικό γράψιμο με την μέθοδο insertMany, καθώς η εισαγωγή των document σειριακά προκαλεί αποτρεπτικούς χρόνους για το map.

Αφότου τελειώσει η μεταφόρτωση του shuffle αρχείου στο mognodb ο Shuffle Writer ενημερώνει το MapStatus, με ένα dummy BlockManagerId, παρομοίως με το Redis. Η διαχείριση του dummy BlockManager γίνεται στον ShuffleBlockFetcherIterator, όπου και σημειώνουμε τα block του mongodb έτσι ώστε να τα προσκομίσουμε. Επίσης, όπως και στο redis.

Αντιστοίχως όπως και στο redis, το mongodb διαχειρίζεται το διάβασμα των δεδομένων με την δημιουργία ενός τρόπου προσκόμισης περισσότερων ή λιγότερων από 5 ταυτόχρονων request, καθώς αυτό είναι το όριο που θέτει το spark από μόνο του. Αυτό επιτυγχάνεται με 2 βασικά στοιχεία:

1. Την συρρίκνωση σε μέγεθος των request, έτσι ώστε να μπορέσουμε προσκομίσουμε περισσότερα request ταυτοχρόνως. Αυτό επιτυγχάνεται στην μέθοδο collectFetchRequests, όπου γίνεται το άθροισμα των block σε request. Το άθροισμα των block γίνεται με βάση τα μεγέθη τους και πιο συγκεκριμένα ένα block έχει μέγιστο κατώφλι από bytes, maxBytesInFlight / 5, το οποίο σημαίνει ότι ταυτόχρονα μπορεί να έχει το πολύ 5 requests προς προσκόμιση. Στην περίπτωση μας θέλουμε να έχουμε περισσότερα ή λιγότερα request προς προσκόμιση ταυτόχρονα, έτσι μικραίνουμε το μέγεθος του request σε targetSize = maxBytesInFlight / SPARK_MONGODB_MAX_REQS_PER_EXECUTOR όπου ο παρανομαστής είναι ο αριθμός των συνδέσεων που ορίζουμε στην αρχή και ορίζεται σύμφωνα με τον πίνακα 3.5.

2. Δημιουργούμε την μέθοδο fetchUpToMaxBytesMongo() η οποία είναι υπεύθυνη για την δουλειά του fetchUpToMaxBytes() για τα request του mongo. Η μέθοδος fetchUpToMaxBytesMongo ελέγχει με βάση την συνθήκη (bytesInFlight == 0 || (reqsInFlight + 1 <= conf.get(SPARK_REDIS_MAX_REQS_PER_EXECUTOR) && bytesInFlight + fetchReqQueue.front.size <= maxBytesInFlight)) αν το request

είναι προσκομήσιμο. Η κύρια διαφορά με την συνθήκη του fetchUpToMaxBytes() είναι πως εδώ ο μέγιστος αριθμός request μπορεί να είναι ίσος με την μεταβλητή SPARK_REDIS_MAX_REQS_PER_EXECUTOR. Αν το request περάσει τον έλεγχο, τότε καλείται η μέθοδος fetchFromMongo για να γίνει ασύγχρονη προσκόμιση των block.

Algorithm 5 fetchUpToMaxBytesMongodb Algorithm

```
dbFetchRequests, queue of requests .
procedure FETCHUPTOMAXBYTESMONGODB
    while ISREMOTEBLOCKFETCHABLE(dbFetchRequests) do
        request ← DEQUEUE(dbFetchRequests)          ▷ Dequeue block of head
        FETCHFROMMONGO(request)
    end while
    procedure ISREMOTEBLOCKFETCHABLE(dbFetchRequests)
        return dbFetchReuests.notEmpty && (dbReqsInFlight + 1 <=
        SPARK_MONGODB_MAX_REQS_PER_EXECUTOR
        bytesInFlight + fetchReqQueue.front.size <= maxBytesInFlight)
    end procedure
end procedure
```

Η μέθοδος fetchFromMongo δημιουργεί ένα thread, έτσι ώστε να προσκομίσει τα ζητούμενα block χωρίς να καθυστερεί το υπόλοιπο πρόγραμμα. Με αυτό αναζητούμε στην συλλογή shuffleMetadata το πεδίο shuffleName, με βάση τα ονόματα των block των request, έτσι ώστε να πάρουμε τα μοναδικά αναγνωριστικά Ids των αντίστοιχων document, για να μπορέσουμε να τα αναζητήσουμε έπειτα στη συλλογή shuffleBlock. Αφού λάβουμε τα document, τα ομαδοποιούμε ανάλογα με το shuffleName ώστε σε περίπτωση που ένα shuffle αρχείο έχει 2 εγγραφές (π.χ. λόγω κάποιας αστοχίας του worker ξανά εκτελέστηκε κάποιο map task, υπάρχει η περίπτωση 2 εγγραφών με το ίδιο shuffleName) στη συλλογή shuffleMetadata,να κρατάμε αυτή με την πιο πρόσφατη ημερομηνία Date. Στην συνέχεια δημιουργείται ένα for-loop έτσι ώστε να προσκομίσει τα block ένα ένα σε σειρά. Αρχικά κάνει ένα query προς τον mongoCluster ζητώντας το block το οποίο έχει metadataId αυτό το οποίο βρήκαμε στην συλλογή shuffleMetadata για shuffleName = ”shuffle_” + shuffleId + ”_” + mapId. Έπειτα αναζητάμε στην συλλογή shuffleBlock με βάση τα πεδία metadataId, reduce όπου reduce=reduceId του συγκεκριμένου block και το πεδίο n το οποίο είναι 1 (όπως αναφέραμε πιο πάνω έχει δημιουργηθεί ένα ευρετήριο index πάνω στα πεδία metadataId, reduce και n για να έχουμε καλύτερη απόδοση ως προς την ανάγνωση). Ξεκινάμε από το πρώτο chunk και έπειτα ελέγχουμε το πεδίο next (το οποίο είναι αληθές σε περίπτωση ύπαρξης επόμενου chunk), έτσι ώστε να το προσκομίσουμε και αυτό. Στην συνέχεια κάνουμε query για την εύρεση του επόμενου block. Και στην περίπτωση αυτή, όπως και του redis τα δεδομένα του test μας χωράνε στην μνήμη και έτσι τα τυλίγουμε (wrap) γύρω από ένα Netty ByteBuf, τον οποίο αναλαμβάνει έπειτα η κλάση NettyManagedBuffer, που χρησιμοποιεί το spark στην περίπτωση που τα προσκομιζόμενα δεδομένα είναι στην μνήμη, για να τα διαχειριστεί στην συνέχεια.

Έχει υλοποιηθεί και η περίπτωση όπου τα δεδομένα μας δεν χωράνε στην μνήμη, παρότι αυτή η περίπτωση δεν έχει ελεγχθεί, καθώς οι έλεγχοι μας έγιναν με μικρά μεγέθη block. Σε αυτήν την περίπτωση αποθηκεύουμε το κάθε λαμβανόμενο κομ-

μάτι σε ένα προσωρινό αρχείο, το οποίο στην συνέχεια διαχειρίζεται από την κλάση FileSegmentManagedBuffer. Το μέγεθος του κομματιού προσκόμισης πρέπει να έχουμε φροντίσει να έχει μέγεθος ίσο με min(set_block_size, 15MB, maxReqSizeShuffleToMem/2). Όπου το set_block_size είναι το ορισμένο μέγεθος block που έχουμε επιλέξει από την μεταβλητή περιβάλλοντος, 15MB καθώς το όριο εγγραφής σε μέγεθος για το mongodb είναι 16MB, και τέλος το maxReqSizeShuffleToMem/2 αποτελεί το μέγιστο αριθμό που μπορούμε να προσκομίσουμε. Ο λόγος που το μέγεθος είναι το μισό του (maxReqSizeShuffleToMem) είναι επειδή, σύμφωνα με την υλοποίηση μας, χρησιμοποιούμε την συνάρτηση getData της κλάσης binary του mongodb η οποία δημιουργεί ένα αντίγραφο των δεδομένων στην μνήμη, και δεν πρέπει να ξεπεραστεί το ορισμένο κατώφλι maxReqSizeShuffleToMem της μνήμης, καθώς την στιγμή της αντιγραφής χρειαζόμαστε τον διπλάσιο χώρο αφού πρέπει να χωρέσει και το πρωτότυπο και το αντίγραφο.

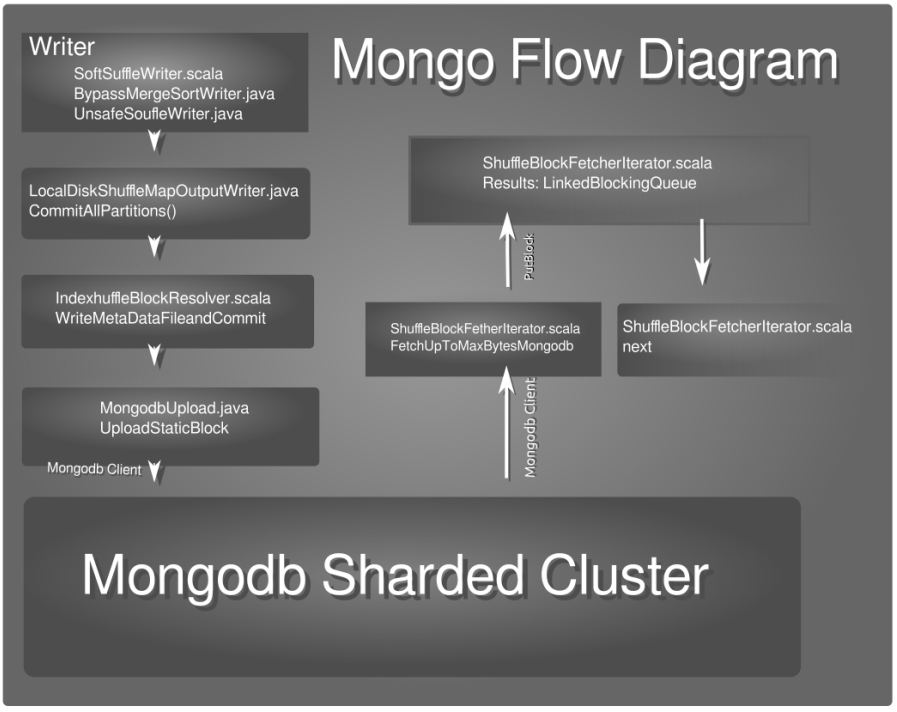
Algorithm 6 fetchFromMongo Algorithm

procedure FETCHFROMMONGO(dbFetchRequest)
  start thread
  get metadataIds for requested blocks from shuffleMetadata Collection
  $BlockMetadata < BlockId, MetadataId > \leftarrow matchblockidswithmetadataids$
  if $dbFetchRequestmaxReqSizeShuffleToMem$ **then**
    for block in dbFetchRequest.blocks **do**
      $curMetadata \leftarrow BlockMetadata(blockId)$
      $reduce \leftarrow block.reduceId$
      $n \leftarrow 1$
      $contvar \leftarrow true$
      while $contvar$ **do**
        $document \leftarrow mongodb.shuffleBlock.find(curMetadata, reduce, n)$
        save document.data to file
        $n \leftarrow n + 1$
        $contvar \leftarrow document.next$
      end while
      update result queue with new block
    end for
  else
    for block in dbFetchRequest.blocks **do**
      $curMetadata \leftarrow BlockMetadata(blockId)$
      $reduce \leftarrow block.reduceId$
      $n \leftarrow 1$
      $contvar \leftarrow true$
      while $contvar$ **do**
        $document \leftarrow mongodb.shuffleBlock.find(curMetadata, reduce, n)$
        save document.data to byteBuf
        $n \leftarrow n + 1$
        $contvar \leftarrow document.next$
      end while
      update result queue with new block
    end for
  end if
end procedure

Όταν στο spark η εφαρμογή φτάνει στο τέλος, ο driver ενημερώνει τον mongodb sharded cluster, έτσι ώστε να διαγράψει (drop) την βάση δεδομένων με όνομα appId, στην οποία βρίσκονται οι συλλογές shuffleBlock και shuffleChecksum για την παραπάνω εφαρμογή.

Τέλος, στο σχήμα 3.11 βλέπουμε συνοπτικά την ροή των δεδομένων του shuffle με το mongodb με βάση τις κλάσεις που καλούνται για την πραγματοποίηση του:

1. Αρχικά ο writer γράφει τα δεδομένα του shuffle σε ένα αρχείο.
2. Όταν τελειώσει η διαδικασία εγγραφής των δεδομένων στο αρχείο καλείται η μέθοδος commitAllPartitions που είναι υπεύθυνη για το τελική υποβολή (commit) του αρχείου.
3. Σε αυτό το σημείο καλείται η μέθοδος uploadStaticBlock της στατικής μεθόδου MongodbUpload η οποία αναλαμβάνει να μεταφορτώσει τα δεδομένα στο mongodb.
4. Έπειτα, κατά την ανάγνωση, καλείται η μέθοδος fetchUpToMaxBytesMongodb η οποία με την σειρά της καλεί την μέθοδο fetchFromMongo για την προσκόμιση του block, το οποίο ενημερώνει την μεταβλητή result, που είναι μια ουρά που διαθέτει όλα τα προσκομισμένα block που δεν έχουν διαβαστεί ακόμα.
5. Τέλος, η μέθοδος next διαβάζει ένα ένα τα block από την result, η οποία ξανακαλεί την μέθοδο fetchUpToMaxBytes και fetchUpToMaxBytesMongodb έτσι ώστε να εκκινήσουν την προσκόμιση του επόμενου block.



Σχήμα 3.11: Διάγραμα ροής κώδικα Mongodb

3.3 Spark, Redis, Mongodb Πακετοποίηση (Containarization) και παράταξη (Deployment)

Όπως αναφέραμε και στο κεφάλαιο 1, χρησιμοποιούμε το kubernetes ως το εργαλείο διαχείρισης του cluster του spark, του redis και του mongodb. Αρχικά, όσο αναφορά το Spark, δημιουργούμε μια εικόνα docker (docker image) η οποία περιέχει τον κώδικα του spark ο οποίος έχει χτιστεί με το εργαλείο mvn[22] ή το sbt[23], το pyspark, τα shell script για εκκίνηση των διαφορετικών ρόλων του spark, δηλαδή worker, master και driver και τέλος το workload, που θα χρησιμοποιήσουμε για τον έλεγχο. Μερικά παραδείγματα υλοποιήσεων docker image για το spark μπορουν να βρεθούν στα [31], [32].

Δημιουργία spark cluster στο kubernetes:

1. Αρχικά δημιουργούμε ένα namespace spark, στο οποίο θα βρίσκονται όλα τα pod του spark.
2. Για την ρύθμιση των master και worker δημιουργούμε ένα configmap αρχείο που περιέχει τις ρυθμίσεις του spark-defaults.
3. Για το master δημιουργούμε deployment αρχείο με 1 replica, με την custom εικόνα που δημιουργήσαμε και ξεκινά με το script του ρόλου master. Έπειτα ανοίγουμε και τις πόρτες 7077 που είναι η προεπιλεγμένη πόρτα του master και την πόρτα 8080 που είναι η πόρτα του UI (διεπαφή χρήστη) του master. Με αυτόν τον τρόπο δημιουργούμε ένα spark-master pod. Παράλληλα, δημιουργούμε και ένα service spark-master στις ίδιες πόρτες για την επικοινωνία του spark master με τα υπόλοιπα spark pods.
4. Για το worker δημιουργούμε deployment αρχείο με n replica, με την custom εικόνα που δημιουργήσαμε και ξεκινά με το script του ρόλου worker. Έπειτα ανοίγουμε και την πόρτα 8081 που είναι η προεπιλεγμένη πόρτα για το UI (διεπαφή χρήστη), με αυτόν τον τρόπο δημιουργούμε τα spark-worker pods. Παράλληλα, δημιουργούμε και ένα service spark-worker για να μπορούμε να έχουμε πρόσβαση στις πληροφορίες της πόρτας.
5. Για το driver δημιουργούμε deployment αρχείο με 1 replica, με την custom εικόνα που δημιουργήσαμε, έπειτα ανοίγουμε και την πόρτα 4040 που είναι η προεπιλεγμένη πόρτα για το UI (διεπαφή χρήστη). Με αυτόν τον τρόπο δημιουργούμε το spark-driver pod. Παράλληλα, δημιουργούμε και ένα service spark-driver για να μπορούμε να έχουμε πρόσβαση στις πληροφορίες της πόρτας.

Δημιουργία redis cluster στο kubernetes:

1. Αρχικά δημιουργούμε ένα namespace redis-cluster στο οποίο θα βρίσκονται όλα τα pod του redis.
2. Δημιουργούμε configmap για τις ρυθμίσεις του redis.

3. Δημιουργούμε ένα statefulset, το οποίο θα έχει m replicas. Το πρώτο container είναι το επίσημο container του redis από το docker hub. Οι πόρτες που ανοίγουμε είναι η 7000 για την σύνδεση με του client, η 17000 για την επικοινωνία μεταξύ των redis κόμβων. Παράλληλα, δημιουργούμε και ένα service redis στις παραπάνω αντίστοιχες πόρτες.

Δημιουργία mongodb cluster στο kubernetes:

1. Αρχικά δημιουργούμε ένα namespace mongodb στο οποίο θα βρίσκονται όλα τα pod του mongodb.
2. Για κάθε shard και config server δημιουργούμε ένα statefulset το οποίο έχει 3 replicas και για εικόνα χρησιμοποιούμε την επίσημη εικόνα του mongo, η πόρτα που ανοίγουμε είναι η 27017 που χρησιμοποιείται για την επικοινωνία. **Στα πλαίσια της εργασίας χρησιμοποιούμε κλασσικά volume αντί για persistent volumes για τα mongo instances, καθώς υποθέτουμε ότι δεν χάνουμε κόμβους και χρησιμοποιούμε το mongo cluster μόνο για το testing του shuffle.** Παράλληλα, δημιουργούμε και ένα service για κάθε shard στην πόρτα 27017 έτσι ώστε να υπάρχει επικοινωνία μεταξύ των pod.
3. Όσο αναφορά το mongos απλά δημιουργούμε ένα replicaset, πάλι με την επίσημη εικόνα του mongo και με πόρτα την 27017. Στο mongos δεν χρειάζεται να δημιουργήσουμε volume καθώς ο mongos λειτουργεί μόνο ως διαμεσολαβητής μεταξύ του client και των shard. Παράλληλα, δημιουργούμε και ένα service για το mongos στην πόρτα 27017 έτσι ώστε να υπάρχει επικοινωνία με τους client.

Κεφάλαιο 4

Αποτίμηση Αποτελεσμάτων

Στο παρόν κεφάλαιο παρουσιάζεται η αποτίμηση των αποτελεσμάτων μας, όπως επίσης και το test-setup και οι μηχανισμοί που χρησιμοποιήθηκαν για την εξαγωγή των μετρικών μας. Πιο συγκεκριμένα στο υποκεφάλαιο 4.1 παρέχουμε τα τεχνικά χαρακτηριστικά των εικονικών μηχανών με τα οποία έγιναν τα test για τις αποτιμήσεις μας, στο υποκεφάλαιο 4.2 παρέχεται το εργαλείο αυτοματοποίησης που χρησιμοποιήθηκε για την ρύθμιση των εικονικών μηχανών, το χτίσιμο και το deployment της εφαρμογής. Παρουσιάζονται βασικές δοκιμές που αφορούν το stress testing του μηχανισμού shuffling 4.3. Έπειτα, στο υποκεφάλαιο 4.4 παρουσιάζονται κάποιες βασικές δοκιμές για το vanilla spark και το redis spark για μεγαλύτερο μέγεθος δεδομένων. Τέλος, παρουσιάζεται η δοκιμή αντοχής στα λάθη από την πλευρά του spark 4.5.

4.1 Setup αποτίμησης αποτελεσμάτων

Αρχικά το setup αποτελείται από 14 VMs (Virtual Machines) όπου κάθε ένα από αυτά αποτελεί ένα κόμβο για το kubernetes cluster. Ένας κόμβος εξ' αυτών αποτελεί τον master του kubernetes cluster και οι υπόλοιποι αποτελούν τους worker κόμβους. Κάθε κόμβος έχει τα εξής χαρακτηριστικά :

1. Επεξεργαστής (CPU) Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (8 vcores)
2. Μέγεθος RAM : 16GB
3. Δίσκος : 80TB

Τέλος, η μεταξύ τους επικοινωνία πραγματοποιείται μέσω διασύνδεσης ethernet στα 10Gbps.

4.2 Εργαλεία αυτοματοποίησης

Όσο το πλήθος των κόμβων του δικτύου που έχουμε να διαχειριστούμε είναι μικρό, η διαχείριση τους είναι σχετικά εύκολη. Όσο όμως μεγαλώνει το πλήθος των κόμβων αυξάνεται εκθετικά η δυσκολία της ρύθμισης (configuration) και της διαχείρισης των κόμβων αυτών. Τα σημερινά cluster μπορούν εύκολα να ξεπεράσουν τους 1000 κόμβους και όπως αντιλαμβανόμαστε η διαχείριση τους με συμβατικά μέσα, (scripts κλπ) είναι πολύ δύσκολη και σχεδόν αδύνατη σε πολλές περιπτώσεις ακόμα και όταν έχουμε να διαχειριστούμε μικρά σχετικά cluster. Γι αυτό το σκοπό έχουν αναπτυχθεί πολλά

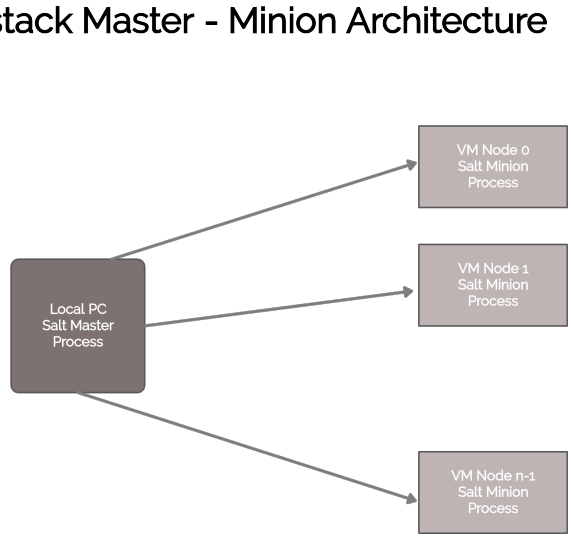
εργαλεία όπως το ansible[33], saltstack[34], jenkins[35] κλπ για τον αυτοματισμό των ρυθμίσεων (configuration) της διαχείρισης και του deployment εφαρμογών.

Στην παρούσα εργασία επιλέξαμε να χρησιμοποιήσουμε το εργαλείο Saltstack. Όπως προαναφέραμε το saltstack είναι ένα εκ των παραπάνω εργαλείων αυτοματοποίησης, το οποίο είναι γρήγορο, έξυπνο και υψηλά κλιμακώσιμο. Έχει γραφτεί σε python και είναι ανοιχτού κώδικα. Για την αυτοματοποίηση των διεργασιών το saltstack χρησιμοποιεί αρχεία YAML συνδυασμένα με event-driven αυτοματοποίηση για την ρύθμιση (configuation) και το deployment περίπλοκων πληροφοριακών συστημάτων[36].

Ενδεικτικά έχουμε χρησιμοποιήσει το salt για μια πληθώρα από λειτουργίες, μερικές εξ'αυτών είναι:

1. Εγκατάσταση του kubernetes
2. Δημιουργία του kubernetes cluster
3. Καταστροφή του kubernetes cluster
4. Ανέβασμα αρχείων
5. κλπ

Το saltstack χρησιμοποιεί την αρχιτεκτονική master-client, στην οποία ο master εκδίδει εντολές προς εκτέλεση στους client. Στο περιβάλλον του salt, το ρόλο του master καταλαμβάνει ο Salt Master, ο οποίος εκδίδει εντολές προς ένα η περισσότερα Salt Minions που έχουν ενσωματωθεί στον συγκεκριμένο master, τα οποία στην ουσία είναι οι client. Ουσιαστικά, το salt μπορεί να περιγράφει ακόμα και ως ένα μοντέλο publisher-subscriber model όπου ο master, εκδίδει εργασίες οι οποίες πρέπει να εκτελεστούν από εγγεγραμμένα Salt Minions[37]. Στην εργασία μας, master αποτελεί ο τοπικός μας υπολογιστής και τα minions αποτελούνται από τα VMs του cluster μας ??.



Σχήμα 4.1: Αρχιτεκτονική Saltstack

Το saltstack παρέχει πολλούς έξυπνους μηχανισμούς, όπως είναι το Salt State, το Salt Pillar και το Salt Orchestration, είναι πολύ βοηθητικά και χρησιμοποιούνται εκτενώς στην εργασία μας. Αρχικά η μέθοδος του Salt States, καθορίζει σε ποια κατάσταση πρέπει να βρίσκεται μέσω κάποιου YAML αρχείου που έχουμε ορίσει εμείς[37]. Ένα YAML αρχείο μπορεί να έχει πολλές καταστάσεις, την μια μετά την άλλη και με αυτόν το τρόπο μπορούμε να βάλουμε σε ουρά πολλές διαφορετικές εργασίες μέσω καταστάσεων. Μια κατάσταση μπορεί να είναι για παράδειγμα η εκτέλεση κάποιων bash εντολών, η κάποια μεταφόρτωση αρχείου από τον master σε κάποιο minion.

Άλλο ένα θεμελιώδες και χρήσιμο χαρακτηριστικό που μας παρέχει το salt, είναι το pillar, το οποίο φυλάσσει απόρρητα και υψηλής διαβάθμισης δεδομένα αποθηκευμένα στον master και τα διαμοιράζει στα minions. Τέτοια δεδομένα μπορεί να είναι usenames και passwords[37]. Αυτά εν προκειμένω είναι αρκετά χρήσιμα, καθώς στον master διαφυλάσσουμε π.χ. τα διαπιστευτήρια του docker τα οποία τα διαμοιράζουμε στα minions. Έκτος από δεδομένα υψηλής εμπιστευτικότητας φυλάσσονται και άλλα δεδομένα όπως π.χ. δεδομένα ρυθμίσεων (configurations), ομάδες (groups) κλπ.

Τέλος, ένα πολύ χρήσιμο χαρακτηριστικό του saltstack είναι το orchestration, που μας παρέχει την δυνατότητα να συντονίσουμε τις διεργασίες πολλαπλών μηχανών από μια κεντρική μονάδα (τον master), και με αυτόν τον τρόπο μπορούμε να ελέγξουμε την σειρά εκτέλεσης διεργασιών. Αυτό είναι πολύ σημαντικό, στην συγκεκριμένη περίπτωση, καθώς πολλές φορές πρέπει να μεταφέρουμε ένα αρχείο από ένα minion στα υπόλοιπα, έτσι ώστε να εκτελεστούν κάποιες εντολές[37]. Χαρακτηριστικό παράδειγμα θα μπορούσε είναι η δημιουργία του kubernetes cluster, όπου ένα minion (kubernetes master) παράγει το connection token για την διασύνδεση των υπόλοιπων, οπότε τα υπόλοιπα minions θα εκτελέσουν τις καταστάσεις τους αφού έχει εξαχθεί το token από το πρώτο minion.

4.3 Μετρήσεις Stress Testing

```
1
2  bytes_per_map_task = size // map_tasks
3  bytes_per_map_per_reduce_task = \
4      bytes_per_map_task // reduce_tasks
5  strings_per_map_per_reduce_task = \
6      bytes_per_map_per_reduce_task // string_length
7  total_strings_per_map = \
8      reduce_tasks*strings_per_map_per_reduce_task
9  total_strings = total_strings_per_map * map_tasks
10
11 rt = reduce_tasks
12 rn = string_length
13     str_num_per_reducer = \
14         sc.parallelize(range(total_strings), map_tasks) \
15         .map(lambda x: (x % rt, (''.join(random.choice(string.
               ascii_uppercase + string.digits) for _ in range(rn)))
               )) \
16         .groupByKey(numPartitions=reduce_tasks) \
```

```
17          .map(lambda x: (x[0],len(x[1]))) \
18          .collect()
```

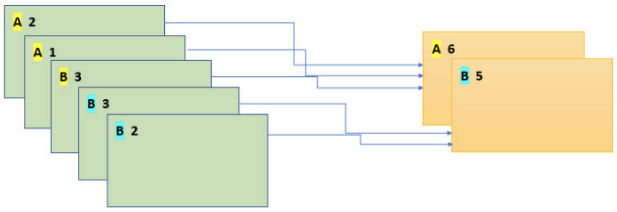
Για την μέτρηση της απόδοσης έγινε με ένα pyspark stress test workload script, εστιασμένο στον μηχανισμό του shuffle, και είναι παρόμοιο με τις δημοσιεύσεις του magnet [9] και του cherry [8]. Ακολουθήθηκε και εφαρμόστηκε η τεχνική της δημοσίευσης του cherry, από όπου έχει και παρθεί ο κώδικας του script [31] με την μικρή αλλαγή στο groupByKey, όπου εδώ αλλάζουμε τον αριθμό των partition έτσι ώστε να αλλάζει ο αριθμός των διεργασιών reduce (reduce task).

Το script αρχικά δέχεται 4 ορίσματα, το πρώτο αναφέρεται στο μέγεθος των δεδομένων τα οποία θα γίνουν shuffle π.χ. 10gb. Το δεύτερο όρισμα αναφέρεται στον βαθμό παραλληλίας των δεδομένων ή με άλλα λόγια πόσα map task θα δημιουργηθούν π.χ. 1000, δηλαδή θα σπάσουμε τα δεδομένα σε 1000 διαφορετικά map task. Το τρίτο όρισμα αναφέρεται στον αριθμό των partition του groupByKey ή με άλλα λόγια ο αριθμός των reduce task ή ο αριθμός των block μέσα σε ένα shuffle αρχείο. Τέλος, το τελευταίο όρισμα είναι υπεύθυνο για το μέγεθος των τυχαίων string που θα δημιουργηθούν ως δεδομένα μας, και σε ένα βαθμό ορίζει και τα granularity καθώς μεγάλο μήκος από string θα είχε ως αποτέλεσμα να μην μπορέσουμε να δημιουργήσουμε το ζητούμενο block size.

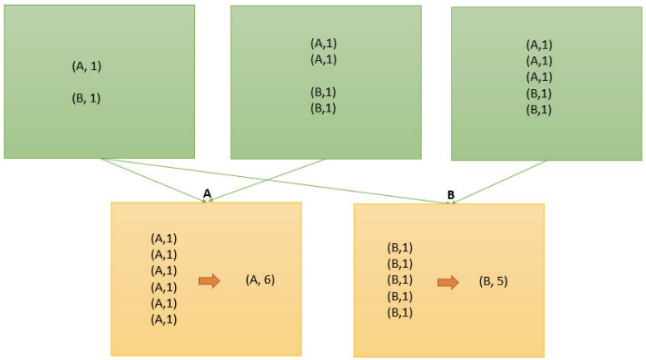
Έπειτα υπολογίζεται τα πόσα string συνολικά θα δημιουργήσουν. Στην συνέχεια δημιουργούμε μια λίστα που έχει συνεχόμενους αριθμούς από 0 μέχρι StringNum-1 και με την συνάρτηση parallelize μοιράζουμε αυτήν την λίστα σε p κομμάτια, όπου p είναι ο αριθμός των map task που έχει δοθεί από το δεύτερο όρισμα. Άρα κάθε map tasks αναλαμβάνει να επεξεργαστεί ένα από αυτά τα κομμάτια των δεδομένων το οποίο έχει εύρος $t_i = [(i-1) * \frac{totalStrings}{mapTasks}, i * \frac{totalStrings}{mapTasks}]$ όπου $i \in [1, p]$. Επομένως, κάθε task με την σειρά του θα αναλάβει να δημιουργήσει $\frac{totalStrings}{mapTasks}$ τυχαία string. Το κάθε task σαρώνει τους αριθμούς στο εύρος t_i που του έχει ανατεθεί και για κάθε ένα δημιουργεί ένα string με k χαρακτήρες, το οποίο ορίζεται ως το τέταρτο όρισμα του script, που μπορεί να είναι τυχαία ψηφία ή κεφαλαία γράμματα και κλειδί του string είναι $j = mod(k, reducePartitions)$ όπου k είναι ο τρέχον αριθμός της λίστας και reducePartitions είναι το τρίτο όρισμα και ο αριθμός των partition που δημιουργεί το groupByKey. Όπως φαίνεται κάθε task δημιουργεί $\frac{\frac{totalStrings}{mapTasks}}{reduceTasks}$ strings για κάθε partition. Άρα αν υποθέσουμε ότι το μέγεθος είναι 1GB και τα mapTasks είναι 100 και τα reduceTask είναι 100 τότε σε κάθε task θα έχουμε $\frac{\frac{1000000000}{100}}{100} = 100$ string για κάθε partition, οπότε το block size στην συγκεκριμένη περίπτωση θα είναι 100KB.

Για την πραγματοποίηση του shuffling χρησιμοποιούμε την συνάρτηση GroupByKey, και ο λόγος που χρησιμοποιούμε την συγκεκριμένη συνάρτηση έναντι οποιαδήποτε άλλης, όπως για παράδειγμα την ReduceByKey, είναι το γεγονός ότι πραγματοποιεί το shuffling των δεδομένων στην αρχή χωρίς τα δεδομένα να υποστούν κάποια περαιτέρω επεξεργασία. Αν πάρουμε για παράδειγμα την ReduceByKey, τα δεδομένα του κάθε partition περνούν από την προεπεξεργασία του reduce τοπικά (στον mapper) και το αποτέλεσμα είναι αυτό που προορίζεται προς το shuffling για τους reducers και έχει ως αποτέλεσμα την μείωση της κίνησης του δικτύου. Αυτό καθιστά ακατάλληλες συναρτήσεις όπως την reduceByKey, καθώς το μέγεθος του block μειώνεται. Εμείς επιθυμούμε να μεταφέρουμε τα δεδομένα στο shuffling απαράλλαχτα και για αυτό,

όπως προαναφέραμε, επιλέγουμε την GroupByKey.



Σχήμα 4.2: ReduceByKey



Σχήμα 4.3: GroupByKey

Σχήμα 4.4: ReduceByKey vs GroupByKey **hahahaha**

Στην συνέχεια, δημιουργούνται pod για το Apache Spark ανάλογα με το ρόλο που τους ανατίθεται (Master, Driver, Worker). Ως αποθηκευτικό μέσο χρησιμοποιείται ο δίσκος του κόμβου στον οποίο έχει τρέχει το pod, για αποθήκευση των παραγόμενων αρχείων του. Σε περίπτωση όπου το pod χαθεί τα παραγόμενα αρχεία και δεδομένα του pod χάνονται μαζί με αυτό.

1. Spark Worker Pod: 1 CPU, 3GB RAM, τοποθεσία: Kubernetes Worker Nodes
2. Spark Master Pod: 1 CPU, 1GB RAM, τοποθεσία: Kubernetes Node for master and driver
3. Spark Driver Pod: 1 CPU, 1GB RAM, τοποθεσία: Kubernetes Node for master and driver

Παρακάτω παρουσιάζονται οι ρυθμίσεις με τις οποίες έγινε το setup του spark, δίνουμε όλους τους πόρους (resources) του worker, σε έναν μόνο executor:

1. spark.executor.cores : 1 πυρήνας
2. spark.executor.memory : 3 Gb

Ρυθμίσεις driver

1. spark.driver.cores : 1 πυρήνας
2. spark.driver.memory : 3 Gb
3. spark.driver.port : 6500
4. spark.driver.blockManager.port : 7500

Οι μετρήσεις πάρθηκαν με τις προαναφερόμενες ρυθμίσεις και για μικρά block sizes. Για μεγαλύτερα μεγέθη, π.χ. 300KB και για μέγεθος δεδομένων 30GB, παρατηρήθηκαν τεράστιοι χρόνοι και από την πλευρά του vanilla spark και από την πλευρά των υλοποιήσεων μας (δοκιμάσαμε μόνο το redis) όσο αναφορά τους χρόνους του reduce, οπότε θα εστιάσουμε μόνο στα μικρά block sizes.

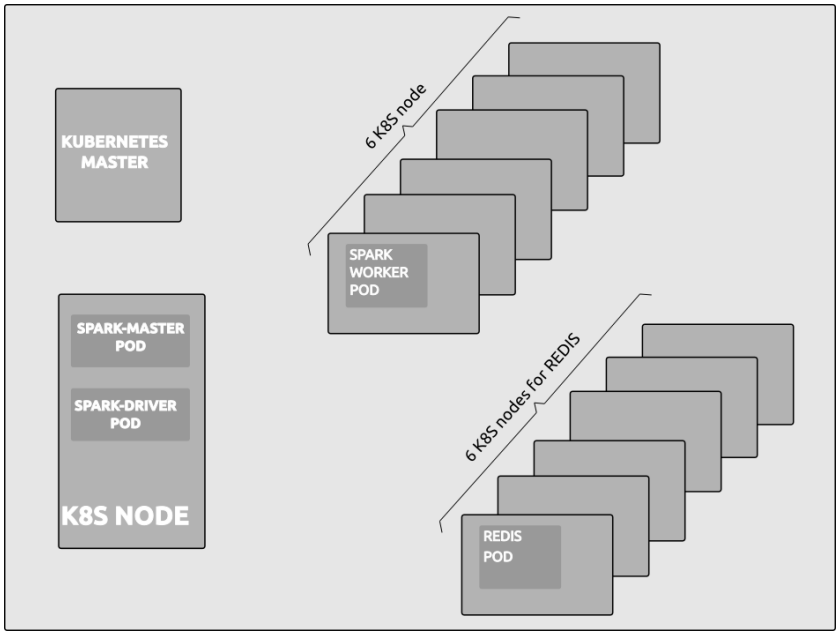
Στον πίνακα 4.1 παρουσιάζονται τα αποτελέσματα για το stress test που πραγματοποιήθηκε στο κλασσικό vanilla spark, χωρίς την χρήση external shuffle service. Από αυτά προκύπτει ότι όσο αναφορά το μέγεθος block 30KB ο χρόνος που πήραμε, τρέχοντας το πείραμα 3 φορές, έχει μια μικρή διακύμανση από 35 μέχρι και 42, το οποίο ίσως να ωφείλεται σε κάποια καθυστέρηση του δικτύου η κάποια καθυστέρηση στο I/O του δίσκου. Στον πίνακα των αποτελεσμάτων παραθέτουμε την καλύτερη περίπτωση με χρόνο reduce 35 sec.

Όσον αναφορά τα block μεγέθους 3KB, παρατηρούμε πως ο χρόνος ολοκλήρωσης του reduce stage, αυξάνεται περίπου 5 φορές. Τρέχοντας το πείραμα 2 φορές, παρατηρούμε χρόνους στο reduce stage 153 και 157. Σε αυτό το σημείο γίνεται εμφανής και η καθυστέρηση I/O του δίσκου, καθώς θα πρέπει να γίνουν πολύ περισσότερες προσβάσεις στον δίσκο για ανάκτηση των ενδιάμεσων δεδομένων του shuffle, όπως επίσης πρέπει να γίνουν και πολύ περισσότερες μεταφορές

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map / Reduce
Vanilla Spark	30GB	1000	1000	30KB	4165 / 35
Vanilla Spark	30GB	1000	10000	3KB	4242 / 153

Πίνακας 4.1: Πίνακας χρόνων για vanilla spark

Παρακάτω παρουσιάζουμε τα αποτελέσματα για την αρχιτεκτονική με το Redis Cluster. Χρησιμοποιήθηκαν, όπως φαίνεται και στο παρακάτω σχήμα, 6 redis pod, ο καθένας σε διαφορετικό κόμβο του kubernetes. Σε κάθε redis pod ανατίθεται 1 cpu και δεν θέτουμε όριο στην μνήμη, ώστε να μπορεί να χρησιμοποιηθεί ολόκληρη η διαθέσιμη μνήμη. Όσο αναφορά τις ρυθμίσεις του redis χρησιμοποιούμε τις προκαθορισμένες και αφαιρούμε το persistence και είναι οι εξής : 1) port 7000 2) cluster-enabled yes 3) cluster-config-file node.conf 4) cluster-node-timeout 5000. Όλοι οι κόμβοι (6) του redis είναι master χωρίς replica.



Σχήμα 4.5: Apache Spark - Redis Kubernetes Deployment

Οι ρυθμίσεις που χρησιμοποιούμε για τον client jedis, από την μεριά του worker, είναι οι εξείς:

1. cfg.setMaxTotal
2. cfg.setMaxIdle
3. cfg.setMinIdle

Οι υπόλοιπες ρυθμίσεις μένουν στις προκαθορισμένες τους τιμές, για παράδειγμα:

1. Η setBlockWhenExhausted (η οποία καθορίζει αν ο client πρέπει να περιμένει, αν οι πόροι του pool εξαντλούνται) μένει στην προκαθορισμένη τιμή (true)
2. Η ρύθμιση για την ανίχνευση ανενεργών συνδέσεων setTimeBetweenEvictionRuns παραμένει στο -1 (δηλαδή ανενεργή)
3. Η ρύθμιση για το μέγιστο χρόνο αναμονής σύνδεσης παραμένει στο -1 (δηλαδή δεν γίνεται ποτέ timeout)
4. Όσο αναφορά τις ρυθμίσεις testOnBorrow, testOnReturn, οι οποίες ελέγχουν τις συνδέσεις πριν ληφθούν και τοποθετηθούν αντιστοίχως στο pool των συνδέσεων, είναι στην προκαθορισμένη τιμή και ανενεργές

Όπως αναφέραμε στο υποκεφάλαιο 3.1.3, το spark δημιουργεί πακέτα με blocks ανάλογα την διεύθυνση. Κάθε πακέτο μπορεί να έχει μέγιστο μέγεθος maxBytesInFlight / 5, πράγμα που σημαίνει ότι στην χειρότερη περίπτωση, θα έχουμε 5 πακέτα προς προσκόμιση, παράλληλα. Εμείς επεκτείναμε αυτήν την λογική κάνοντας την σύμβαση πως κάθε πακέτο θα έχει μέγεθος maxBytesInFlight / redisMaxReqsPerExecutor

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map / Reduce
Redis 5 connections no pipeline	30GB	1000	1000	30KB	4346 / 46
Redis 5 connections no pipeline	30GB	1000	10000	3KB	4746 / 203
Redis 10 connections no pipeline	30GB	1000	1000	30KB	4343 / 46
Redis 10 connections no pipeline	30GB	1000	10000	3KB	4819 / 285

Πίνακας 4.2: Πίνακας χρόνων για redis - Apache Spark - no pipeline - maxTotal = maxIdle = minIdle

όπου redisMaxReqsPerExecutor (SPARK_REDIS_MAX_REQS_PER_EXECUTOR) είναι ο αριθμός των αδρανών συνδέσεων σε κάθε κόμβο του cluster (δηλαδή ο κάθε κόμβος του redis έχει redisMaxReqsPerExecutor συνδέσεις), έτσι ώστε να έχουμε την δυνατότητα να προσκομίσουμε περισσότερα πακέτα παράλληλα.

Σε όλα τα παραδείγματα χρησιμοποιούμε της προκαθορισμένες τιμές για τις μεταβλητές maxBytesInFlight και maxReqSizeShuffleToMem, οι οποίες είναι 48MB και 200MB αντιστοίχως.

Όπως αναφέραμε, η προκαθορισμένη τιμή για το maxBytesInFlight είναι 48MB, άρα μπορούμε να προσκομίζουμε παράλληλα 48MB / numConnection, στην περίπτωση που έχουμε ορίσει το numConnections να είναι redisMaxReqsPerExecutor. Αν το redisMaxReqsPerExecutor είναι 10 τότε, παράλληλα, μπορούμε να έχουμε 10 πακέτα των 4.8MB. Στην περίπτωση στην οποία έχουμε 1000 map tasks και 1000 reduce tasks, το κάθε reducer task προσκομίζει 30MB (1000 map tasks * 30KB block size). Άρα στο reduce θα δημιουργηθούν $\lceil \frac{30MB}{4.8MB} \rceil = 7$ πακέτα, που μπορούν να εκτελεστούν παράλληλα. Στην περίπτωση όπου έχουμε: 10 connectionNum, 1000 διεργασίες map (map task) και 10000 διεργασίες reduce (reduce task), τότε κάθε reduce task θα προσκομίζει 3MB, άρα τώρα όλα θα πάνε σε ένα πακέτο. Στην περίπτωση των 100 συνδέσεων, θα μπορούμε να έχουμε, παράλληλα, 100 πακέτα των 480KB. Άρα στην πρώτη περίπτωση όπου map tasks = 1000 και reduce task = 1000, δημιουργούνται 63 πακέτα και για map task = 1000 και reduce task = 10000 δημιουργούνται 7 πακέτα.

Για τις μετρήσεις στον πίνακα 4.2 θέτουμε τις παρακάτω ρυθμίσεις για τον jedis client του κάθε worker: cfg.setMaxTotal = redisMaxReqsPerExecutor, cfg.setMaxIdle = redisMaxReqsPerExecutor και cfg.setMinIdle = redisMaxReqsPerExecutor, όπου το redisMaxReqsPerExecutor, όπως αναφέραμε, είναι η μεταβλητή περιβάλλοντος που ορίσαμε SPARK_REDIS_MAX_REQS_PER_EXECUTOR. Σε αυτήν την περίπτωση δεν ενεργοποιούμε το pipeline.

Παρατηρούμε πως ο χρόνος της μεταφόρτωσης μεγαλώνει, όσο μεγαλώνουν και τα block του κάθε map, καθώς πρέπει να γίνουν περισσότερες μεταφορτώσεις. Βλέπουμε πως η παράλληλη προσκόμιση ενός - ενός των block από πολλά νήματα (thread) παράλληλα, δεν μας δίνει καλύτερη απόδοση από το παραδοσιακό spark. Παρατηρούμε επίσης πως οι περισσότερες συνδέσεις δεν μας δίνουν καλύτερα αποτελέσματα στο reduce.

Στην συνέχεια προχωράμε στην μελέτη της απόδοσης του spark με την μέθοδο του pipeline για το διάβασμα των block στο στάδιο του reduce. Όπως αναφέραμε, η συγκεκριμένη μέθοδος εφαρμόζεται σε μικρού μεγέθους blocks και έχει ως αποτέλεσμα να γράφονται πάντα στην μνήμη, επειδή ένα request δεν πρόκειται να ξεπεράσει ποτέ το μέγεθος του maxReqSizeShuffleToMem, (το μικρό μέγεθος block παρέχει καλό granularity). Στην συνέχεια κάνουμε ελέγχους για ρυθμίσεις με cfg.setMaxTotal = redisMaxReqsPerExecutor, cfg.setMaxIdle = redisMaxReqsPerExecutor και cfg.setMinIdle = redisMaxReqsPerExecutor

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map / Reduce
Redis 1 connection pipeline	30GB	1000	1000	30KB	4337 / 36
Redis 1 connection pipeline	30GB	1000	10000	3KB	4756 / 107
Redis 5 connections pipeline	30GB	1000	1000	30KB	4315 / 40
Redis 5 connections pipeline	30GB	1000	10000	3KB	4822 / 144

Πίνακας 4.3: Πίνακας χρόνων για redis - Apache Spark - pipeline - maxTotal = maxIdle = minIdle

Παρατηρούμε πως, στην περίπτωση του pipeline, για πολύ μικρό αριθμό block πετυχαίνουμε ένα ταχύτερο αποτέλεσμα στο reduce, σε σχέση με το κλασσικό spark. Αυτό έχει να κάνει με την φύση του pipeline που, όπως αναφέραμε, δεν χρειάζεται να περιμένει για απάντηση από κάθε block. Με αυτό κερδίζει πολύ χρόνο, όπως φαίνεται και από τα αποτελέσματα. Προφανώς, ένας από του λόγους που πετυχαίνει καλύτερα αποτελέσματα, είναι και το γεγονός ότι όλα τα δεδομένα βρίσκονται στην μνήμη και δεν χρειάζεται να γίνει πρόσβαση στον δίσκο. Άλλη μια σημαντική παρατήρηση, μπορεί να είναι πως, όπως και στην περίπτωση της μη χρήσης του pipeline, τα αποτελέσματα με περισσότερες συνδέσεις δεν φέρουν και καλύτερο αποτέλεσμα. Αυτό είναι εμφανές και στην παρούσα περίπτωση, όπου βλέπουμε πως μία (1) σύνδεση αποδίδει καλύτερα αποτελέσματα σε σχέση με τις πέντε (5).

Όπου 1 σύνδεση σημαίνει πως θα προσπαθήσει το πρόγραμμα να χωρέσει όσα

περισσότερα block μπορεί σε ένα request αντι γία περισσότερα. Άρα θα προσπαθήσει να δημιουργήσει πάντα ένα reqiest με μέγεθος το πολύ maxBytesInFlight και αφού τελειώσει το request τότε ξεκινάει το επόμενο request.

Στην συνέχεια στον πίνακα 4.4 γίνονται έλεγχοι για τις ρυθμίσεις cfg.setMaxTotal = redisMaxReqsPerExecutor, cfg.setMaxIdle = redisMaxReqsPerExecutor και cfg.setMinIdle = 1, όπου κρατάμε μόνο μια αδρανή σύνδεση προς κάθε κόμβο. Σε αυτήν την περίπτωση ενεργοποιούμε το pipeline.

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map / Reduce
Redis 5 connections pipeline	30GB	1000	1000	30KB	4338 / 52
Redis 5 connections pipeline	30GB	1000	10000	3KB	4787 / 164

Πίνακας 4.4: Πίνακας χρόνων για redis - Apache Spark - pipeline - maxTotal = maxIdle = minIdle - 5 connections - 1 min idle

Εδώ παρατηρούμε, ότι η αρχική δημιουργία μιας μόνο σύνδεσης, έχει αντίκτυπο και στο αποτέλεσμα του reduce stage. Συγκρίνοντας το αποτέλεσμα του πίνακα 4.5 και του πίνακα 4.4 παρατηρούμε πως η αρχικοποίηση των αδρανών συνδέσεων σε 1, επιβραδύνει αρκετά το αποτέλεσμα, ακόμα και σε περίπτωση pipeline, με αποτέλεσμα χρόνους βραδύτερους από αυτούς του κανονικού spark.

Στη συνέχεια παραθέτουμε κάποιες παρατηρήσεις από την χρήση του redis για το shuffle των δεδομένων του spark:

1. Έχουμε κάνει την υπόθεση πως τα ενδιάμεσα δεδομένα του spark χωράνε πάντα στην μνήμη του redis και δεν έχουμε εξετάσει την περίπτωση στην οποία γεμίζει η μνήμη κάποιου κόμβου. Σε αυτή την περίπτωση, κανονικά, το redis χρησιμοποιεί την swap μνήμη του συστήματος, έτσι ώστε να μπορέσει να αποθηκεύσει τα επιπλέον δεδομένα, αυτό θα είχε ως αποτέλεσμα την επιβράδυνση του συστήματος. Όμως το kubernetes, στην επίσημη σελίδα, προτείνει [38] την απενεργοποίηση της swap μνήμης από όλους τους κόμβους. Πειραματικά, μπορεί να γίνει η χρήση swap, θέτοντας το flag fail-on-swap σε false κατά την έναρξη του kubelet. Παράλληλα, στην καινούργια έκδοση kubernetes v1.22 προσφέρεται μια πρώιμη υλοποίηση χρήσης swap, που θα μπορούσε να δοκιμαστεί σε μελλοντικές υλοποιήσεις [39].

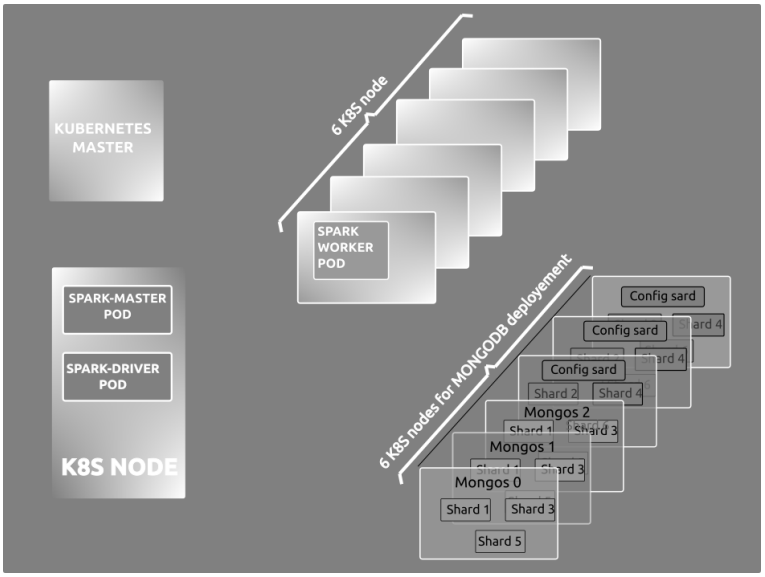
2. Μια δεύτερη παρατήρηση είναι πως αν ο αριθμός των application και συνεπώς ο αριθμός των executor αυξηθεί, υπό την προϋπόθεση ότι ο αριθμός των κόμβων του redis cluster παραμένει σταθερός, τότε ίσως να παρατηρηθεί μια πτώση της απόδοσης του redis, καθώς ο αριθμός των συνδέσεων αυξάνεται. Όπως παρατηρήσαμε και στις δοκιμές μας. Σε περιπτώσεις όπου υπάρχουν πολλές συνδέσεις,

ίσως είναι προτιμότερο να κλιμακωθεί με περισσότερους κόμβους το redis.

Στην συνέχεια παραθέτουμε τις εξαγόμενες μετρικές από τα παραδείγματα μας με το Mongodb. Και σε αυτήν την περίπτωση χρησιμοποιήθηκε η ίδια παράταξη (setup) των spark κόμβων. Όσο αναφορά το mongodb δημιουργήθηκαν 6 shard τα οποία χρησιμοποιούν 6 kubernetes κόμβους για την λειτουργία τους. Σε αυτό το σημείο θέτουμε σε κάθε shard τον κανόνα πως δεν πρέπει 2 αντίγραφα (replica) του stateful set, του συγκεκριμένου shard, να βρίσκονται στον ίδιο kubernetes κόμβο, αλλά και ανάλογα το shard τοποθετούμε τα pod σε συγκεκριμένους κόμβους όπως βλέπουμε παρακάτω. Έτσι με αυτόν τον κανόνα πετυχαίνουμε ένα διαμοιρασμό, όπου σε ένα kubernetes κόμβο θα βρίσκονται αντίγραφα (replica) διαφορετικών shard. Παράλληλα δημιουργούμε και ένα διακομιστή ρυθμίσεων (config server) για την λειτουργία του sharded cluster. Τέλος, δημιουργούμε και 3 mongoS τα οποία εξυπηρετούν την διεπαφή μεταξύ του sharded cluster και του client. Και σε αυτήν την περίπτωση, τα mongoS βρίσκονται μέσα σε 3 συγκεκριμένους από τους 6 κόμβους που έχει παραταχθεί το sharded cluster, αλλά και πάλι διαφορετικά αντίγραφα (replicas) του mongos βρίσκονται σε διαφορετικούς kubernetes κόμβους.

Πόροι και setup κάθε pod:

1. Shard i pod: 1 CPU, 3GB RAM, δημιουργία τοπικού emptyDir volume στον κόμβο του kubernetes για την αποθήκευση των δεδομένων, όπου εαν το i mod 2 = 0 τότε τα shard κατατάσσονται στα mongodb kubernetes nodes VM1, VM2 και VM3 αλλιώς στα mongodb kubernetes nodes VM4, VM5, VM6
2. Config server pod: 1 CPU, 1GB RAM. Δημιουργία τοπικού emptyDir volume στον κόμβο του kubernetes για την αποθήκευση των δεδομένων, τοποθεσία mongodb kubernetes nodes VM4, VM5, VM6.
3. mongos Pod: 1 CPU, 1GB RAM, τοποθεσία mongodb kubernetes nodes VM1, VM2, VM3.



Σχήμα 4.6: Apache Spark - Mongodb Kubernetes Deployment

Ο client, από την πλευρά του worker, δημιουργείται με τις ακόλουθες ρυθμίσεις:
maxSize(conf.get(config.SPARK_MONGODB_MAX_REQS_PER_EXECUTOR))
minSize(conf.get(config.SPARK_MONGODB_MAX_REQS_PER_EXECUTOR)) (ή
minSize(1) ανάλογα με τις ρυθμίσεις)
maxConnectionIdleTime(120000, TimeUnit.MILLISECONDS), δημιουργούμε ένα σταθερό αριθμό από συνδέσεις και ο τρόπος δημιουργίας των πακέτων είναι πανομοιότυπος, όπως περιγράφτηκε πιο πάνω, με το redis, με την αλλαγή όπου το redisMaxReqsPerExecutor γίνεται mongodbMaxReqsPerExecutor, (SPARK_MONGODB_MAX_REQS_PER_EXECUTOR). Όσο αναφορά της υπόλοιπες ρυθμίσεις όπως είναι για παράδειγμα το write concertn, έχουν παραμείνει default.

Όσο αναφορά τον τρόπο μεταφόρτωσης των δεδομένων, από τον worker στο cluster του mongodb, επιλέχθηκε η εισαγωγή των δεδομένων να γίνει μαζικά, με την συνάρτηση insertMany. Καθώς οι χρόνοι στην περίπτωση ¨ένα κάθε φορά¨ ήταν αποτρεπτικοί. Π.χ. για 6GB δεδομένων παρατηρήθηκε, από ένα διαφορετικό πείραμα, πως ο χρόνος ήταν 5588 για ένα την φορά και μαζικά ήταν 1698. Στην παρούσα περίπτωση ο κάθε worker επιλέγει τον mongoS στον οποίο θα συνδεθεί σύμφωνα με το executorId και πιο συγκεκριμένα σύμφωνα με τον τύπο executorId mod mongoSArray.length, όπου mongoSArray είναι ο πίνακας με τις διευθύνσεις των mongoS. Ο driver πάντα συνδέεται στον 0.

Όσο αναφορά το chunk size του κάθε block, επιλέγουμε ένα μεγάλο μέγεθος και πιο συγκεκριμένα 261120 έτσι ώστε να μην δημιουργηθούν πολλαπλά chunk για το ίδιο block καθώς τα μεγέθη μας είναι 3KB και 30KB.

Από τα αποτελέσματα στον πίνακα 4.5 παρατηρούμε πως οι χρόνοι του mongodb είναι αρκετά μεγαλύτεροι σε σχέση με αυτούς του redis και vanilla spark. Μια παρατήρηση που μπορεί να γίνει είναι πως, ειδικότερα κατά την διάρκεια του map, οι καθυστερήσεις (latency) δεν είναι σταθερές από task σε task όπως φαίνεται και στην εικόνα 4.11. Αυτό έχει και ως αποτέλεσμα οι χρόνοι να μην είναι τόσο σταθεροί από εκτέλεση σε εκτέλεση. Επίσης η ενδιάμεση επικοινωνία (proxy) που δημιουργείται μεταξύ του client και sharded cluster, μέσω του mongoS, πιστεύουμε πως σίγουρα προσθέτει ένα overhead στα αποτελέσματα μας. Μελετώντας την χρήση της CPU μέσα από το εργαλείο παρακολούθησης prometheus δεν παρατηρείται κάποιος φόρτος όσο αναφορά τους mongoS που αναλαμβάνουν την διεπαφή μεταξύ των client και του mongodb sharded cluster, οπότε εικάζουμε πως το πρόβλημα της καθυστέρησης βρίσκεται στην μεριά των shard pod κατά την εγγραφή των δεδομένων. Για την καλύτερη απόδοση της χρήσης του mongodb απαιτείται διερεύνηση και αρκετό fine-tuning όπως και επίσης ενδεχόμενες αλλαγές στις σχεδιαστικές επιλογές της αρχιτεκτονικής μας, όπως και της παράταξης των ρυθμίσεων των pod πάνω στο kubernetes.

Υποθέτουμε πως ένας από τους παράγοντες της καθυστέρησης της εισόδου είναι το γεγονός πως κάθε εισαγωγή στην συλλογή έχει ως αποτέλεσμα και την ενημέρωση του index [40], μια άλλη αιτία **εικάζουμε** πως μπορεί να είναι και ο διαγωνισμός (contention) μεταξύ των pod για να γράψουν στον δίσκο κάτι που ίσως προκαλεί αυτές τις καθυστερήσεις. Άλλο ένα σημείο που μπορούμε να παρατηρήσουμε είναι πως ο χρόνος του σταδίου του map, όταν τα reduce task είναι 10000, είναι πολύ μεγαλύτερος, καθώς οι εγγραφές είναι πολύ περισσότερες και κατά συνέπεια και το overhead του index θα είναι μεγαλύτερο.

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map / Reduce
Mongodb 5 connections, batch size 1000000	30GB	1000	1000	30KB	7660/196 7835/212 10612/235
Mongodb 5 connections min idle connections 1, batch size 1000000	30GB	1000	1000	30KB	9461/203 10429/242
Mongodb 5 connections, batch size 1000000	30GB	1000	10000	3KB	11970/1208 14676/1459
Mongodb 5 connections, batch size 5000000	30GB	1000	1000	30KB	8663/222
Mongodb 10 connections, batch size 1000000	30GB	1000	1000	30KB	8212/249 10126/294
Mongodb 10 connections, batch size 1000000	30GB	1000	10000	3KB	12258/1641

Πίνακας 4.5: Πίνακας χρόνων για mongodb - Apache Spark

Όσο αναφορά τους χρόνους βλέπουμε πως δεν υπάρχει σχετικά σταθερός χρόνος, καθώς όπως αναφέραμε, οι καθυστερήσεις μεταξύ των διεργασιών δεν είναι σταθερές. Μια παρατήρηση που έγινε σε αυτό το σημείο είναι πως υπάρχει υποβιβασμός της απόδοσης του mongo όσο τρέχουμε περισσότερα παραδείγματα. Δηλαδή, για παράδειγμα σε πρώτο χρόνο τρέξαμε το mongodb με 5 συνδέσεις και reduce tasks = 10000, βλέπουμε πως είχαμε 14676s για το map και 1459s για το reduce. Σε δεύτερο χρόνο τρέχοντας πάλι το ίδιο πείραμα, σε μια διαφορετική κατάταξη του mongo, πήραμε αποτέλεσμα 11970s για το map και 1208s για το reduce. Βλέπουμε πως οι χρόνοι δεν

είναι σταθεροί.

Όσο αναφορά το διάβασμα, βλέπουμε πως έχουμε σχετικά σταθερά αποτελέσματα, τα οποία είναι πάλι πολύ μεγαλύτερα από το redis και το vanilla κάτι που είναι αναμενόμενο καθώς σε αυτήν την περίπτωση δεν έχει υλοποιηθεί μαζική ανάγνωση των block, αλλά γίνεται 1-1 με την σειρά όπως και στην πρώτη περίπτωση του redis. Όσο αναφορά τις διαφορετικές ρυθμίσεις παρατηρούμε ότι ούτε η αύξηση του batch size, ούτε ο αριθμός των ελάχιστων συνδέσεων, δεν συμβάλουν στην βελτίωση του αποτελέσματος.

4.4 Μετρήσεις Vanilla Spark και Redis Spark με 60GB

Όσο αναφορά τα 60GB πήραμε μερικές μετρήσεις για το vanilla spark, όπως επίσης και για το την καλύτερη περίπτωση που καταφέραμε να βρούμε στο spark με το redis, η οποία ορίζεται ως μια σύνδεση προς κάθε κόμβο και ανάγνωση με τεχνικές pipeline.

Τρέξαμε το stress test script για μέγεθος δεδομένων 60GB με 1500 map tasks και 10000 reduce tasks, το οποίο μας δίνει ως αποτέλεσμα ένα block size 4KB. Τρέχοντας το vanilla spark πήραμε χρόνους για το map περίπου στα 8332 - 8369 δευτερόλεπτα, όσο αναφορά το reduce πήραμε αποτελέσματα από 204 - 216 δευτερόλεπτα. Στην συνέχεια τρέξαμε το ίδιο πείραμα με το redis, ορισμένο με τις προαναφερόμενες ρυθμίσεις και για το map πήραμε χρόνου από 9176 - 9200 δευτερόλεπτα, κάτι που είναι αναμενόμενο, καθώς, όπως αναφέραμε, η μεταφόρτωση πολλών block έχει αντίκτυπο στις επιδόσεις του map. Στο reduce πήραμε χρόνους 142, 145, 150 και 177 δευτερόλεπτα, παρατηρούμε πως μπορούμε να πετύχουμε ταχύτερους χρόνους σε σχέση με το vanilla spark (χωρίς ESS). Σε μια περίπτωση το redis στο reduce μας έδωσε και ένα outlier στα 215 δευτερόλεπτα, παρ' όλα αυτά επανεκκινώντας το kubernetes cluster επανήλθαμε στους κανονικούς χρόνους στα 145 δευτερόλεπτα.

4.5 Αντοχή Στα Λάθη από την πλευρά του Spark Worker

Για να μπορέσουμε να επεκτείνουμε την υλοποίηση μας, έτσι ώστε να μην επαναϋπολογίζονται τα χαμένα map task, σε περίπτωση που είναι ενεργοποιημένο το shuffle με το redis ή το mongodb, χρειάστηκε να παρέμβουμε στις συναρτήσεις οι οποίες χειρίζονται αυτήν την λειτουργία. Η πρώτη κλάση που χρειάζεται αλλαγή είναι η TaskSetManager και πιο συγκεκριμένα η μέθοδος executorLost, η οποία ενεργοποιείται την ώρα που χάνεται ένας executor. Σε αυτή την συνάρτηση υπάρχει ο έλεγχος αν πρέπει να επαναϋπολογιστούν τα map task του χαμένου executor. Σε αυτό το σημείο, λοιπόν, ελέγχουμε αν είναι ενεργοποιημένο το redis ή map shuffle, αν ναι δεν τις επαναϋπολογίζουμε. Το δεύτερο σημείο βρίσκεται στην κλάση DAGScheduler και πιο συγκεκριμένα στην συνάρτηση handleExecutorLost, όπου προσθέτουμε, επίσης, τον έλεγχο για το redis και το mongodb, έτσι ώστε να μην υπάρξει επαναϋπολογισμός. Από τα παραπάνω αποτελέσματα είναι φανερό πως το mongodb χρήζει βελτίωσης, όπως αναφέραμε, ως προς την αρχιτεκτονική και ως προς το fine tuning, έτσι ώστε να πετύχουμε πιο σταθερά και πιο γρήγορα αποτελέσματα.

```
// also check if shuffle or mongodb shuffle are activated in that case dont resubmit lost tasks
if (isShuffleMapTasks && !env.blockManager.externalShuffleServiceEnabled && !isZombie
&& !conf.get(config.SPARK_REDIS_SHUFFLE_ENABLE) &&
  !conf.get(config.SPARK_MONGODB_SHUFFLE_ENABLE)) {
```

Σχήμα 4.7: TaskSetManager executor lost

```
// extracheck for redis and mongodb if they are enabled dont recalculate map tasks
val fileLost = ((workerHost.isDefined || !env.blockManager.externalShuffleServiceEnabled)
  && !RedisServerEnv.shuffleEnabled && !MongodbServerEnv.shuffleEnabled)
```

Σχήμα 4.8: DAGScheduler executor lost

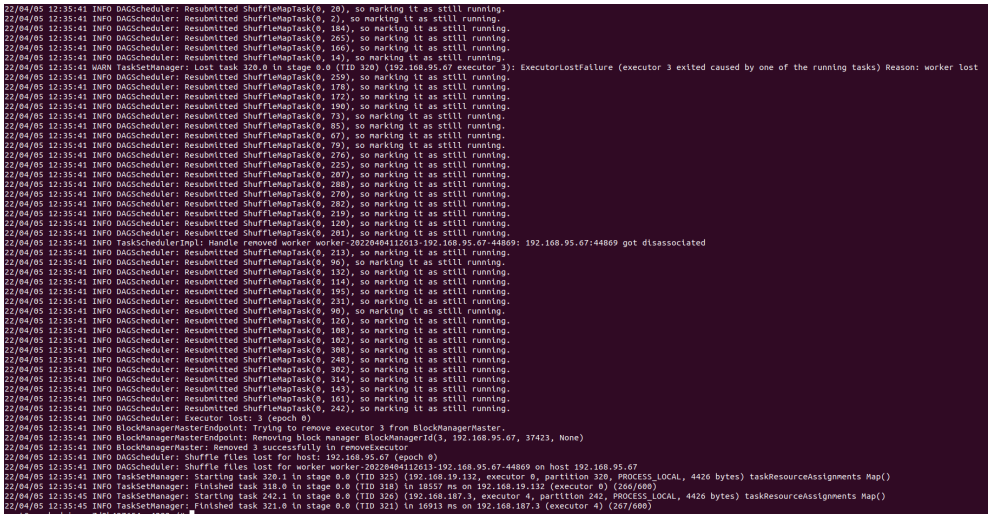
Στο παρόν υποκεφάλαιο θα εξετάσουμε την περίπτωση αστοχίας ενός τυχαίου worker κόμβου, κατά την διάρκεια του map σε διάφορα ποσοστά ολοκλήρωσης του map. Παρομοίως με την δημοσίευση [8]. Προσομοιώνουμε την αστοχία σκοτώνοντας ένα pod του worker deployment, από το kubectl (command line tool για το kubernetes). Αυτό έχει ως αποτέλεσμα να επανεκκινηθεί το worker pod και έτσι τα shuffle δεδομένα του παλιού χάνονται. Ως test χρησιμοποιήσαμε το stress test script με 12GB από δεδομένα και 600 map tasks και 1000 reduce tasks. Τα αποτελέσματα φαίνονται στον πίνακα για το vanilla spark, το redis και το mongodb.

Type	Data Size	Map Tasks	Reduce Tasks (blocks in shuffle file)	Block Size	Time for Map	Failure Percentage
Vanilla Spark	12GB	600	1000	20KB	1707	0%
Vanilla Spark	12GB	600	1000	20KB	1778	25%
Vanilla Spark	12GB	600	1000	20KB	1876	50%
Vanilla Spark	12GB	600	1000	20KB	1938	75%
Redis	12GB	600	1000	20KB	1739	0%
Redis	12GB	600	1000	20KB	1753	25%
Redis	12GB	600	1000	20KB	1758	50%
Redis	12GB	600	1000	20KB	1744	75%
Mongodb	12GB	600	1000	20KB	3848	0%
Mongodb	12GB	600	1000	20KB	3979	25%
Mongodb	12GB	600	1000	20KB	3902	50%
Mongodb	12GB	600	1000	20KB	3759	75%

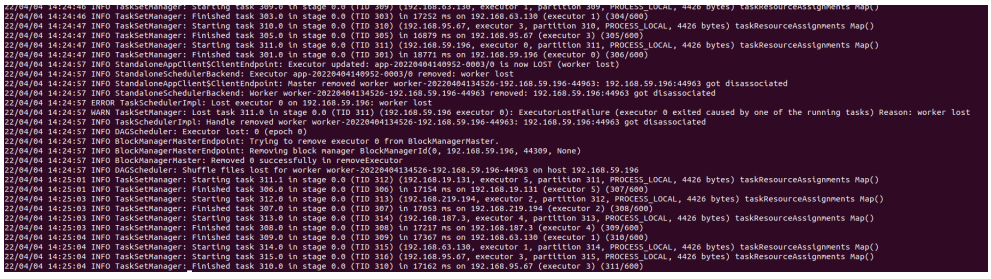
Πίνακας 4.6: Πίνακας failure tolerance

”Σκοτώνουμε” ένα worker pod περίπου στο 25%, δηλαδή όταν έχουν ολοκληρωθεί περίπου 150 map tasks, στο 50%, δηλαδή όταν έχουν ολοκληρωθεί περίπου 300 map tasks και στο 75%, δηλαδή όταν έχουν ολοκληρωθεί περίπου 450 map tasks.

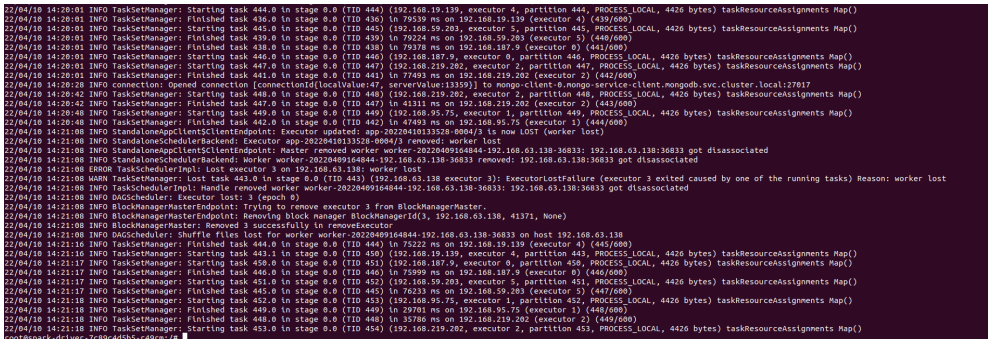
Παρατηρούμε πως στο vanilla spark, όσο πιο αργά χαθεί ο worker τόσα περισσότερα αποτελέσματα map task χάνονται και για αυτό ο χρόνος του map μεγαλώνει. Όσο αναφορά το redis βλέπουμε πως δεν υπάρχει αύξηση στο χρόνο του map, καθώς δεν χάνονται δεδομένα. Όσο αναφορά το mongodb γενικότερα υπάρχει μια διακύμανση στα αποτελέσματα του map, η οποία δεν ήταν από κάποιον επαναϋπολογισμό χαμένου task καθώς εδώ, όπως έχουμε αναφέρει, είναι όλα αποπλεγμένα από τους worker, αυτή προέρχεται λόγω των μεταβαλλόμενων latency και των μη σταθερών χρόνων γενικότερα, η οποία δεν φαίνεται να είναι τόσο μεγάλη όσο του πίνακα 4.5. Παρ’ όλα αυτά τρέχοντας σε διαφορετικό χρόνο το πείραμα, αφού έχουμε τρέξει και άλλα πειράματα, παίρνουμε διαφορετικούς χρόνους, πάλι της τάξης των 4500s.



Σχήμα 4.9: vanilla spark re-calculate tasks



Σχήμα 4.10: redis-spark doesn’t re-calculate tasks



Σχήμα 4.11: mongodb-spark doesn't re-calculate tasks

Κεφάλαιο 5

Μελλοντική δουλειά

Με την εργασία μας καταφέραμε να επεκτείνουμε το apache spark, αποπλέκοντας το shuffle service. Για τον σκοπό αυτό χρησιμοποιήσαμε το redis και το mongodb. Παρ' όλα αυτά, καθώς το θέμα αποτελεί ένα πολύ ευρύ πεδίο, υπάρχουν πολλά θέματα που δεν έχουν μελετηθεί όπως:

1. Η εξέταση της περίπτωσης όπου υπάρχει σφάλμα από την πλευρά της βάσης. Όπως αναφέραμε στην εργασία υποθέσαμε πως δεν χάνονται δεδομένα, παρ' όλα αυτά σε μια πραγματική εφαρμογή το σφάλμα είναι πολύ πιθανό και μπορούμε να χάσουμε δεδομένα, με άλλα λόγια δεν έχει εξεταστεί η περίπτωση όπου μπορεί να υπάρξει κάποιο fetch failure (σφάλμα που παρουσιάζεται όταν δεν μπορούμε να προσκομίσουμε ένα block), όπως επίσης δεν έχει εξεταστεί η συμπεριφορά του database client με το spark σε περίπτωση αστοχίας. Αν και οι client των βάσεων είναι ένα επίπεδο (layer) στα οποία γίνεται διαχείριση των αστοχιών, δεν έχει εξεταστεί από εμάς ποια θα είναι η συμπεριφορά του συστήματος σε περίπτωση σφάλματος της βάσης.
2. Η υλοποίηση έχει εξεταστεί γιά δεδομένα μέχρι 30GB και για block sizes 3KB και 30KB για όλες τις υλοποιήσεις και 60GB για τις υλοποιήσεις του vanilla και του redis, στο μέλλον μπορεί να γίνει εξέταση για μεγαλύτερα μεγέθη, όπως και επίλυση των προβλημάτων για εξέταση block μεγαλύτερου μεγέθους.
3. Όπως αναφέραμε και στο προηγούμενο κεφάλαιο 4.3, δεν έχει εξεταστεί η περίπτωση όπου εξαντλείται η μνήμη ενός κόμβου redis και καταφύγει στην χρήση μνήμης swap. Πράγμα που θα είχε ως συνέπεια την επιβράδυνση συστήματος. Σε μελλοντικές υλοποιήσεις θα είχε μεγάλο ενδιαφέρον να διερευνηθεί.
4. Εξέταση και βελτίωση του σταδίου του map του redis με τεχνικές pipeline που θα μπορούσαν να βελτιώσουν την απόδοση ακόμα περισσότερο.
5. Ενδιαφέρον θα είχε η εξέταση την υλοποίησης του redis και με άλλες βιβλιοθήκες client που μπορεί να είναι και ασύγχρονες, όπως το lettuce, που **ίσως** να μας πρόσφερε καλύτερη απόδοση, καθώς δεν χρειάζεται να περιμένουν όλα τα blocks, όπως κάνουμε τώρα στο pipeline καθώς το jedis είναι σύγχρονο.
6. Fine-tuning της υλοποίησης του mongodb, επίλυση bugs όπως και αναθεωρήσεις της προτεινόμενης αρχιτεκτονικής, για παράδειγμα μια τροποποίηση, που ίσως να είχε βάση, θα ήταν η δημιουργία ενός τοπικού mongoS για κάθε worker, έτσι ώστε να μην υπάρχει επιπλέον καθυστέρηση στην επικοινωνία), ώστε να πετύχουμε μεγαλύτερη απόδοση.
7. Άλλο ένα ενδιαφέρον πεδίο έρευνας θα αποτελούσε και η επέκταση του shuffle κάποιου άλλου εργαλείου κατανεμημένης ανάλυσης, όπως το flink[5], το hadoop[4] με κάποια από τις υλοποιήσεις μας ή κάποια άλλη κατανεμημένη βάση η custom αρχιτεκτονική.
8. Ένα ενδιαφέρον σημείο έρευνα θα μπορούσε να αποτελέσει και η επέκταση του redis με άλλες κατανεμημένες βάσεις όπως είναι για παράδειγμα το Apache Ignite[41]

Κεφάλαιο 6

Συμπεράσματα

Το Apache Spark αποτελεί ένα πολύ διάσημο εργαλείο για την επεξεργασία δεδομένων μεγάλης κλίμακας. Δημιουργήθηκε για να κάλυψει τις σημερινές ανάγκες στο διαδίκτυο, όπου ο όγκος των δεδομένων αυξάνεται με εκθετικούς ρυθμούς, καθώς και οι απαιτήσεις για άμεση απάντηση. Αυτό έχει ως αποτέλεσμα αρχιτεκτονικές μεμονωμένων μηχανών να είναι ακατάλληλες. Ένα κομμάτι των κατανεμημένων εργαλείων επεξερασίας δεδομένων, το οποίο παρουσιάζει πολύ μεγάλο ενδιαφέρον για έρευνα, όχι μόνο στο apache spark αλλά και σε όλα τα κατανεμημένα εργαλεία ανάλυσης, αποτελεί το κομμάτι του shuffle και έχει αποτελέσει πεδίο μελέτης πολλών ερευνών, καθώς προκαλεί συμφόρηση αλλά και κινδύνους, καθώς ένα σφάλμα σε κάποιον worker μπορεί να προκαλέσει τεράστιους χρόνους για επαναϋπολογισμό των χαμένων δεδομένων.

Στην παρούσα εργασία, έστω και με περιορισμούς, καταφέραμε να αποπλέξουμε τους worker του apache spark, από την φύλαξη και διαχείριση των δεδομένων του shuffle με κατανεμημένες βάσεις δεδομένων, καθιστώντας την ύπαρξη τους εφήμερη και τον ρόλος τους μόνο υπολογιστικό.

Όπως είδαμε το redis, με τεχνικές pipeline, πετυχαίνει το μέγιστο throughput του και με λίγες συνδέσεις καταφέρνει να πετύχει καλύτερα αποτελέσματα από το vanilla spark, για τις συγκεκριμένες ρυθμίσεις και παραμέτρους για μικρά block size. Όσο αναφορά το mongodb είδαμε πως αναμενόμενα η απόδοση του σε σχέση με το redis είναι χειρότερη, καθώς το redis αποθηκεύει τα δεδομένα στην μνήμη (τουλάχιστον στην περίπτωση μας όπου τα δεδομένα χωράνε στην μνήμη). Η απόδοση της παρούσας υλοποίησης είναι χειρότερη και από το vanilla spark παρ' όλα αυτά πιστεύουμε πως με fine-tuning μπορούμε να πάρουμε καλύτερα αποτελέσματα στο μέλλον.

Κατάλογος σχημάτων

Κατάλογος πινάκων

Βιβλιογραφία

[1] D. J. Borkovich και P. D. Noah, «Big data in the information age: Exploring the intellectual foundation of communication theory», *Information Systems Education Journal (ISEDJ)*, *January 2014*, σ. 16, διεύθν.: https://files.eric.ed.gov/fulltext/EJ1140800.pdf.

[2] D. Reinsel, J. Gantz και J. Rydning, «The digitization of the world from edge to core», σ. 6, διεύθν.: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[3] M. Zaharia, R. S. Xin, P. Wendell κ.ά., «Apache spark: A unified engine for big data processing», διεύθν.: https://cs.stanford.edu/~matei/papers/2016/cacm_apache_spark.pdf.

[4] *Hadoop*, https://hadoop.apache.org/, Last Accessed: 2022-03-31.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi και K. Tzoumas, «Apache flink: Stream and batch processing in a single engine», διεύθν.: http://asterios.katsifodimos.com/assets/publications/flink-deb.pdf.

[6] R. Sethi, M. Traverso, D. Sundstrom κ.ά., «Presto: Sql on everything», διεύθν.: https://trino.io/Presto_SQL_on_Everything.pdf.

[7] J. Dean και S. Ghemawat, «Mapreduce: Simplified data processing on large clusters», διεύθν.: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[8] N. Nikitas, I. Konstantinou, V. Kalogeraki και N. Koziris, «Cherry: A distributed task-aware shuffle service for serverless analytics», *2021 IEEE International Conference on Big Data*, διεύθν.: http://www.cslab.ntua.gr/~ikons/BigD469.pdf.

[9] M. Shen, Y. Zhou και C. Singh, «Magnet: Push-based shuffle service for large-scale data processing», διεύθν.: http://www.vldb.org/pvldb/vol13/p3382-shen.pdf.

[10] M. Zaharia, M. Chowdhury, T. Das κ.ά., «Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing», διεύθν.: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[11] *RDD Lineage — Logical Execution Plan*, https://mallikarjuna_g.gitbooks.io/spark/content/spark-rdd-lineage.html, Last Accessed: 2022-03-31.

[12] P. Pedamkar, *Introduction To Spark Shuffle*, https://www.educba.com/spark-shuffle/, Last Accessed: 2022-03-31.

[13] *Redis*, https://redis.io/, Last Accessed: 2022-03-31.

[14] *Redis Persistence*, https://redis.io/topics/persistence, Last Accessed: 2022-03-31.

[15] T. Wickham, *What Redis Deployment Do You Need ?*, https://blog.octo.com/en/what-Redis-deployment-do-you-need/, Last Accessed: 2022-03-31.

[16] *Redis Cluster*, https://redis.io/topics/cluster-tutorial, Last Accessed: 2022-03-31.

[17] *What is a Document Database ?*, https://www.mongodb.com/document-databases, Last Accessed: 2022-03-31.

[18] *MongoDB ReplicaSet*, https://docs.mongodb.com/manual/replication/, Last Accessed: 2022-03-31.

[19] *Docker Overview*, https://docs.docker.com/get-started/overview/, Last Accessed: 2022-03-31.

[20] *What is Kubernetes ?*, https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, Last Accessed: 2022-03-31.

[21] *Apache Spark Github*, https://github.com/apache/spark, Last Accessed: 2022-03-31.

[22] *Apache Maven*, https://maven.apache.org/, Last Accessed: 2022-03-31.

[23] *sbt - The interactive building tool*, https://www.scala-sbt.org/, Last Accessed: 2022-03-31.

[24] *jedis*, https://github.com/redis/jedis, Last Accessed: 2022-03-31.

[25] *Map Status*, https://books.japila.pl/apache-spark-internals/scheduler/MapStatus/#location, Last Accessed: 2022-03-31.

[26] *redis pipeline*, https://redis.io/topics/pipelining, Last Accessed: 2022-03-31.

[27] *Jedis remove by pattern*, https://stackoverflow.com/questions/21317501/redis-jedis-delete-by-pattern, Last Accessed: 2022-03-31.

[28] *Mongodb Sharded Key*, https://docs.mongodb.com/manual/core/sharding-shard-key/, Last Accessed: 2022-03-31.

[29] *Mongodb Indexes*, https://docs.mongodb.com/manual/indexes/, Last Accessed: 2022-03-31.

[30] *GridFS*, https://docs.mongodb.com/manual/core/gridfs/, Last Accessed: 2022-03-31.

[31] N. Nikitas, *Cherry Code*, https://github.com/nikoshet/spark-cherry-shuffle-service, Last Accessed: 2022-03-31.

[32] *Spark Docker Example*, https://github.com/testdrivenio/spark-kubernetes/blob/master/docker/Dockerfile, Last Accessed: 2022-03-31.

[33] *ansible*, https://www.ansible.com/, Last Accessed: 2022-03-31.

[34] *saltstack*, https://saltproject.io/, Last Accessed: 2022-03-31.

[35] *Jenkins*, https://www.jenkins.io/, Last Accessed: 2022-03-31.

[36] *about saltstack*, https://docs.saltproject.io/en/latest/topics/about_salt_project.html#about-saltstack, Last Accessed: 2022-03-31.

[37] *Salt System Architecture*, https://docs.saltproject.io/en/latest/topics/salt_system_architecture.html, Last Accessed: 2022-03-31.

[38] *Kubernetes Setup*, https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin, Last Accessed: 2022-03-31.

[39] E. Hashman, *Kubernetes Alpha Swap Support*, https://kubernetes.io/blog/2021/08/09/run-nodes-with-swap-alpha/, Last Accessed: 2022-03-31.

[40] *Mongodb Write Performance*, https://www.mongodb.com/docs/manual/core/write-performance/, Last Accessed: 2022-03-31.

[41] *Apache Ignite*, https://ignite.apache.org/, Last Accessed: 2022-03-31.