ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ
ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ευφυές Σύστημα Επεξεργασίας Metadata σε Cloud Τεχνολογίες

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ Ε. ΚΑΨΟΥΛΗΣ

Επιβλέπων: Θεοδώρα Βαρβαρίγου

Καθηγήτρια Ε.Μ.Π.

ΑΘΗΝΑ, ΔΕΚΕΜΒΡΙΟΣ 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ
ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ευφυές Σύστημα Επεξεργασίας Metadata σε Cloud Τεχνολογίες

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ Ε. ΚΑΨΟΥΛΗΣ

Επιβλέπων: Θεοδώρα Βαρβαρίγου

Καθηγήτρια Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Δεκεμβρίου 2017.

………………	………………	………………
(Υπογραφή)	(Υπογραφή)	(Υπογραφή)

Αθήνα, Δεκέμβριος 2017

……………….

Νικόλαος Ε. Καψούλης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Νικόλαος Ε. Καψούλης, 2017.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Distributed Knowledge and Media Systems Group του Τομέα Επικοινωνιών, Ηλεκτρονικής και Συστημάτων Πληροφορικής της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη της Καθηγήτριας Θεοδώρας Βαρβαρίγου.

Θα ήθελα να ευχαριστήσω θερμά την κ. Θεοδώρα Βαρβαρίγου, που μου έδωσε την ευκαιρία να ασχοληθώ με ενδιαφέρουσες τεχνολογίες αιχμής, καθώς και για τις πολύτιμες απαντήσεις στις ερωτήσεις μου.

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον Υποψήφιο Διδάκτορα Ε.Μ.Π. Βρεττό Μουλό για την βοήθεια, την καθοδήγηση, την υποστήριξη και την επιμονή του, καθώς και τους συνεργάτες του στο εργαστήριο, Αχιλλέα Μαρινάκη, Γιώργο Χατζηκυριάκο και Πέτρο Κληροδέτη, όντες πάντα πρόθυμοι να απαντήσουν στις απορίες μου.

Η παρούσα εργασία αφιερώνεται στην οικογένειά μου.

Νικόλαος Ε. Καψούλης,
Αθήνα, 13η Δεκεμβρίου 2017

Περίληψη

Σκοπός αυτής της διπλωματικής εργασίας είναι η μελέτη των αλληλοσυσχετίσεων (interrelationships) μεταξύ χαρακτηριστικών συστημάτων αρχείων (filesystems) και με ποιον τρόπο το ένα επηρεάζει το άλλο. Για κάθε file system που μελετήσαμε χωρίσαμε τα χαρακτηριστικά του σε δύο κατηγορίες, εν ονόματι technical και business, και καταλήξαμε στην άμεση αλληλεπίδραση που συμβαίνει μεταξύ τους και στην υπεροχή των technical ως σημαντικότερα. Αυτό είναι και ο βασικός πυλώνας όπου στηρίχθηκε η παρούσα εργασία. Κατασκευάσαμε ένα σύστημα όπου οι πάροχοι θα διαθέτουν προς πώληση τα προϊόντα τους – προϊόντα τύπου file system storage, διαθέτοντας μόνο το όνομα του filesystem και τα τεχνικά του χαρακτηριστικά, χωρίς τα χαρακτηριστικά τύπου business. Τα business θα κρίνονται και θα αποφασίζονται από το ίδιο το σύστημα. Στη συνέχεια, οι πελάτες θα τα αγοράζουν ως Filesystem-As-A-Service Cloud υπηρεσία και το ίδιο το σύστημα θα συντονίζει και θα κάνει τις κατάλληλες προτάσεις στους πελάτες. Όλα τα παραπάνω, τα υλοποιήσαμε στα πλαίσια του συνδυασμού τεχνολογιών Cloud Computing και Big Data, χρησιμοποιώντας και ενώνοντας αντίστοιχες πλατφόρμες. Χρησιμοποιήσαμε το πρότυπο TOSCA Simple Profile YAML version 1.1, τις αναπαραστάσεις αρχείου YAML και JSON, τη μη σχεσιακή βάση δεδομένων MongoDB και τη σύγχρονη υπολογιστική μηχανή Apache Spark.

Λέξεις-κλειδιά: interrelationships, technical, business, technical attribute, business attribute, filesystem, file system, TOSCA Simple Profile YAML version 1.1, YAML, JSON, document-oriented, NoSQL, MongoDB, Sharded Cluster, Apache Spark

Abstract

The purpose of this diploma dissertation, is to study interrelationships between filesystems' attributes and how these impact each other. We divided each filesystem's attributes into two categories: technical and business. We decided that the technical attributes are more important than the business attributes of a filesystem. This is the core argument of the dissertation. We constructed a system where a provider is registering one or more products of filesystem storage type to the system. The provider can only register the name and the technical attributes, not the business ones. The business attributes are decided through system's resolution. Then, customers can buy these Filesystem-As-A-Service type products through our platform. All of the above are deployed and developed through Cloud Computing and Big Data cutting edge technologies compilation. We used the spec TOSCA Simple Profile YAML version 1.1, YAML and JSON representations, the NoSQL database MongoDB and the lightning-fast cluster computing engine Apache Spark.

Keywords: interrelationships, technical, business, technical attribute, business attribute, filesystem, file system, TOSCA Simple Profile YAML version 1.1, YAML, JSON, document-oriented, NoSQL, MongoDB, Sharded Cluster, Apache Spark

Πίνακας Περιεχομένων

Εισαγωγή......14
Κεφάλαιο 1......16
Big Data......16
Εισαγωγή......16
Ορισμός......17
Χαρακτηριστικά Big Data......18
Ευκαιρίες των Big Data......21
Cloud Computing......22
Εισαγωγή......22
Ορισμός......22
Essential Characteristics......23
Service Models......24
Deployment Models......26
Κεφάλαιο 2......27
Βάσεις Δεδομένων......27
Εισαγωγή......27
Σχεσιακό Σύστημα Διαχείρισης Βάσεις Δεδομένων – RDBMS......28
NoSQL Βάσεις Δεδομένων......29
Είδη NoSQL βάσεων δεδομένων......30
Σύγκριση Σχεσιακών και NoSQL Βάσεων......31
ACID......33
Θεώρημα CAP......35
Η βάση δεδομένων MongoDB......36
Εισαγωγή......36
Σημαντικά χαρακτηριστικά της βάσης MongoDB......37
Apache Spark: Lightning-Fast Cluster Computing......40
Εισαγωγή......40
Ιστορική αναδρομή......40
Ανάλυση Αρχιτεκτονικής Δομής......41
Spark Core......43
RDD – Resilient Distributed Datasets......43
Σύγκριση με Hadoop MapReduce......43
Lazy Evaluation......45
Διαχειριστές συμπλέγματος (Cluster Managers)......49
Κεφάλαιο 3......54
Εισαγωγή......54
Top-bottom αρχιτεκτονική του συστήματος......54
Ανάλυση file system attributes......56
Χρήση του προτύπου TOSCA Simple Profile in YAMLv1.1......58
Χρήση της αναπαράστασης JSON......62
Χρήση της MongoDB......64
BSON......64
Sharding......65
Εισαγωγή προϊόντων στο σύστημα......66
Παράδειγμα αποτίμησης προϊόντος......67
Αλληλοσυσχετίσεις technical και business......68
Apache Spark......70

Αρχιτεκτονική components....71
Κεφάλαιο 4....72
Εισαγωγή....72
Πρώτη Μεθοδολογία....72
Δεύτερη Μεθοδολογία....75
Generator....75
Συντακτικός Αναλυτής....77
Πελάτης....77
Interrelationship Scala Object....78
Apache Spark και MongoDB....79
ClusterSparkContext....79
Κεφάλαιο 5....83
Εισαγωγή....83
Simple Technical....84
Simple Business....85
Only Technical 1....86
Only Technical 2....87
Only Business 1....88
Only Business 2....88
Technical & Business without conflict....89
Technical & Business with conflict 1....90
Technical & Business with conflict 2....91
Κεφάλαιο 6....93
Εισαγωγή....93
Συγκριτικά Αποτελέσματα....93
Επαναχρησιμοποίηση....96
Έκβαση του πειράματος....97
Κεφάλαιο 7....97
Εισαγωγή....97
Μελλοντικό Έργο....98
Επεκτάσεις και Βελτιστοποιήσεις....98
Παράρτημα 1....100
Οι τρεις πιθανές κατηγοριοποιήσεις των metadata....100
TOSCA Simple Profile in YAML Version 1.1....101
MongoDB Aggregation Pipeline....103
Apache Spark AccumulatorV2....104
Παράρτημα 2....108
Παράδειγμα JSON αρχείου με 3 filesystem προϊόντα....108
Filesystem Generator....108
Συντακτικός αναλυτής....112
Η εφαρμογή....113
Interrelationships....116
Configuration του SparkContext....118
Παράρτημα 3....119
build.sbt....119
getRandom.sh....121
Cosine Similarity....122
Παράρτημα 4....125
Αυτοματοποίηση σύνδεσης με ssh....125
Εγκατάσταση Apache Spark from scratch....126

Configuration files του Spark......127
Εισαγωγή Προϊόντων στη βάση......132
Υποβολή εφαρμογής στο Spark......133
Βιβλιογραφία – Αναφορές......136

Πίνακας Εικόνων

Εικόνα 1: Big Data στη σύγχρονη εποχή από διαφορετικές πηγές και για διαφορετικούς σκοπούς. 17
Εικόνα 2: Ο όγκος των Big Data παγκοσμίως......18
Εικόνα 3: What happens online in 60 seconds......19
Εικόνα 4: The NIST definition of Cloud Computing......22
Εικόνα 5: Τα 3 Μοντέλα Υπηρεσιών......25
Εικόνα 6: Relational DBMS......28
Εικόνα 7: NoSQL VS RDBMS......33
Εικόνα 8: Big Data VS Structured Data (of RDBMSs)......34
Εικόνα 9: Θεώρημα CAP......35
Εικόνα 10: MongoDB......35
Εικόνα 11: MongoDB και RDBMS Models......36
Εικόνα 12: BSON MongoDB document......36
Εικόνα 13: Αρχιτεκτονική MongoDB Sharded Cluster......39
Εικόνα 14: Apache Spark Logo, Components and APIs......41
Εικόνα 15: Apache Spark: πάνω από 100 φορές πιο γρήγορο από το Hadoop MapReduce......42
Εικόνα 16: Επαναληπτική διαδικασία στο Hadoop MapReduce......44
Εικόνα 17: Διαδραστική διαδικασία στο Hadoop MapReduce......44
Εικόνα 18: Το Spark κάνει ανάκτηση δεδομένων σε ταχύτητα RAM......45
Εικόνα 19: Διαδραστική διαδικασία στο Spark......45
Εικόνα 20: Spark Lazy Evaluation......47
Εικόνα 21: Αρχιτεκτονική του Spark SQL......48
Εικόνα 22: Αρχιτεκτονική του Spark Streaming......49
Εικόνα 23: Apache Spark με Hadoop YARN......51
Εικόνα 24: Αρχιτεκτονική Spark Cluster......52
Εικόνα 25: Παράδειγμα γραφικού περιβάλλοντος Standalone Scheduler......53
Εικόνα 26: Βασικά components της αρχιτεκτονικής του συστήματος......55
Εικόνα 27: Αλληλοσυσχετίσεις – Interrelationships......57
Εικόνα 28: Επέκταση του TOSCA BlockStorage κόμβου......61
Εικόνα 29: Η συλλογή products κατανέμεται στα 3 shards......66
Εικόνα 30: Επίλυση αλληλοσυσχετίσεων......70
Εικόνα 31: Αρχιτεκτονική Components......72
Εικόνα 32: Τύπος υπολογισμού Cosine Similarity......73
Εικόνα 33: Metadata Categories......104
Εικόνα 34: Simple Software Installation (MySQL) on a TOSCA Compute Node......105
Εικόνα 35: Aggregation Pipeline......106
Εικόνα 36: Spark Accumulator......108

Πίνακας Μετρήσεων και Διαγραμμάτων

Πίνακας 1: Βασικές έννοιες ενός Cluster....53
Πίνακας 2: Cosine Similarity προϊόντων 1....74
Πίνακας 3: Cosine Similarity προϊόντων 2....75
Πίνακες 4 και 5: Πίνακες ονομάτων filesystem και τεχνικών χαρακτηριστικών με αποτίμηση....77
Πίνακας 6: Simple Technical Query....85
Πίνακας 7: Simple Business Query....86
Πίνακας 8: Only Technical 1 Query....87
Πίνακας 9: Only Technical 2 Query....88
Πίνακας 10: Only Business 1 Query....89
Πίνακας 11: Only Business 2 Query....90
Πίνακας 12: Technical & Business without conflict Query....91
Πίνακας 13: Technical & Business with conflict 1 Query....92
Πίνακας 14: Technical & Business with conflict 2 Query....93
Πίνακας 15: Συγκριτικό Διάγραμμα των Queries....96
Πίνακες 16 και 17: Πίνακες των μετρήσεων....97

Εισαγωγή

Στην παρούσα διπλωματική εργασία υλοποιήσαμε ένα σύστημα που εξυπηρετεί αιτήματα πελάτη. Τα αιτήματα αυτά σχετίζονται την εύρεση του κατάλληλου Filesystem As A Service προϊόντος για τον πελάτη. Στη σύγχρονη εποχή, το Cloud Computing και η ετερογένεια των υπηρεσιών που προσφέρει, έχουν ανοίξει μία νέα διαδικτυακή αγορά λεγόμενη Everything As A Service. Με την εξέλιξη της τεχνολογίας και την πρόοδο της επιστήμης των υπολογιστών, ο καθένας μπορεί να απολαμβάνει διαφορετικού είδους υπηρεσίες χωρίς να διαθέτει τις υποδομές ή τα κατάλληλα λογισμικά.

Το Υπολογιστικό Νέφος αναπτύσεται πλέον από κάθε είδος εταιρίας, οργανισμού ή συνόλου ατόμων. Πανεπιστήμια, κυβερνήσεις και εταιρίες έχουν ανάγκη τη χρησιμοποίηση των δυνατοτήτων που προσφέρει και τα προβλήματα που λύνει.

Το Cloud Computing σχεδιάζεται και επιλύει σημαντικά σύγχρονα προβλήματα με μεγαλύτερο αυτό των μεγάλων δεδομένων (Big Data). Οι υψηλές αποδόσεις και οι ταχύτητες επεξεργασίας και ανάλυσης δεδομένων σε μικρό χρονικό διάστημα καθιστά αυτές τις δύο έννοιες αλληλένδετες, πράγμα που διαπιστώνεται μέσα σε αυτήν την εργασία.

Η παρούσα διπλωματική εργασία οργανώνεται στα παρακάτω κομμάτια:

- Στο Κεφάλαιο 1 περιγράφονται τα Big Data και το Cloud Computing στη σύγχρονη εποχή έχοντας σαν γνώμονα την ετερογένεια των υπηρεσιών και τη δυναμικότητα των αρχείων που είναι αναγκαία για πολλές εφαρμογές είτε υπηρεσίες.
- Στο Κεφάλαιο 2 περιγράφονται τα Big Data και τρόποι βελτιστοποίησης και επίλυσης των προβλημάτων τους.
- Στο Κεφάλαιο 3 περιγράφεται αναλυτικά το πρόβλημα και αναλύεται η αρχιτεκτονική του συστήματος και τα κομμάτια που τη συναποτελούν.
- Στο Κεφάλαιο 4 αναλύονται δύο διαφορετικές προσεγγίσεις της λύσης του προβλήματος και περιγράφεται και εξηγείται ο κώδικας.
- Στο Κεφάλαιο 5 περιγράφονται τα διαγράμματα και οι μετρήσεις της απόδοσης της πλατφόρμας που κατασκευάστηκε.

- Στο Κεφάλαιο 6 περιγράφονται συγκριτικά αποτελέσματα και συμπεράσματα, αναφέρονται πιθανά κομμάτια του συστήματος προς επαναχρησιμοποίηση και σχολιάζεται η έκβαση του πειράματος.
- Στο Κεφάλαιο 7 περιγράφεται το πιθανό μελλοντικό έργο, επεκτάσεις και βελτιστοποιήσεις του συστήματος και γίνεται αναφορά για χρήση του μοντέλου λύσης σε άλλες κατηγορίες προβλημάτων.

1

Big Data

Εισαγωγή

Στη σύγχρονη εποχή, παρατηρείται καθημερινή παραγωγή μεγάλων δεδομένων από διαφορετικές πηγές και για διαφορετικούς σκοπούς. Επιχειρήσεις, πανεπιστήμια, κυβερνήσεις αλλά και κάθε άτομο προσωπικά παράγουν μεγάλα δεδομένα ή Big Data σε καθημερινή βάση για να καλύψουν τις ανάγκες τους. Αυτή η τεράστια και συχνή παραγωγή δημιουργεί ανάγκες και προκλήσεις για αποθήκευση, ανάλυση, αναζήτηση, κοινοποίηση, μεταφορά, ενημέρωση και προστασία όλων αυτών των δεδομένων.

Εικόνα 1: Big Data στη σύγχρονη εποχή από διαφορετικές πηγές και για διαφορετικούς σκοπούς.

Η πληθώρα και οι υψηλές απαιτήσεις των προκλήσεων αυτών δεν μπορούν να καλυφθούν από τα παραδοσιακά μέσα επεξεργασίας και διαχείρισης δομημένων δεδομένων (structured data) , όπως για παράδειγμα οι σχεσιακές βάσεις δεδομένων, τα υπολογιστικά φύλλα (spreadsheets) και τα data warehouses. Γι' αυτό το λόγο αναζωπυρώθηκαν τεχνολογίες (για παράδειγμα οι NoSQL βάσεις δεδομένων) κατάλληλες για την αποδοτική και γρήγορη επεξεργασία και διαχείριση μεγάλων ημι-δομημένων ή μη-δομημένων δεδομένων (semi-structured ή unstructured data). Από τα βασικότερα χαρακτηριστικά αυτών των νέων τεχνολογιών είναι η υποστήριξη οριζόντιας κλιμάκωσης (horizontal scaling) η οποία τις έχει καταστήσει κατάλληλες για περιβάλλοντα μεγάλων δεδομένων.

Ορισμός

Έχουν δωθεί κατά καιρούς αρκετοί ορισμοί που προσπαθούν να προσεγγίσουν την έννοια των Μεγάλων Δεδομένων ή αλλιώς Big Data. Ο ορισμός της Wikipedia[29] ορίζει τα Big Data σαν σύνολα δεδομένων μεγέθους ακατάλληλου για τις ικανότητες επεξεργασίας, διαχείρισης, αποθήκευσης κ.ά. που προσφέρουν τα κοινώς διαδεδομένα λογισμικά. Ακόμη, η Gartner ορίζει τα Big Data σαν μεγάλου όγκου, μεγάλης ταχύτητας και μεγάλης ποικιλίας πληροφορία που απαιτεί οικονομικούς, καινοτόμους τρόπους επεξεργασίας οι οποίοι βελτιώνουν και ενισχύουν την κατανόηση, την λήψη αποφάσεων και την αυτοματοποίηση. Γενικά, όταν μιλάμε για Big Data κάνουμε λόγο για δεδομένα πολύ μεγάλου μεγέθους, διαφορετικού περιεγχομένου και σκοπού, τα οποία παράγονται ασταμάτητα, με αυξανόμενο ρυθμό και όγκο με αποτέλεσμα οι απαιτήσεις για έλεγχο πάνω τους να ανεβαίνουν συνεχώς. Πρόσφατη έρευνα ισχυρίζεται ότι το 2025 ο ετήσιος αριθμός παραγωγής δεδομένων θα ισούται με 163 Zettabytes[32], δηλαδή περίπου 163 τρισεκατομμύρια Gigabytes.

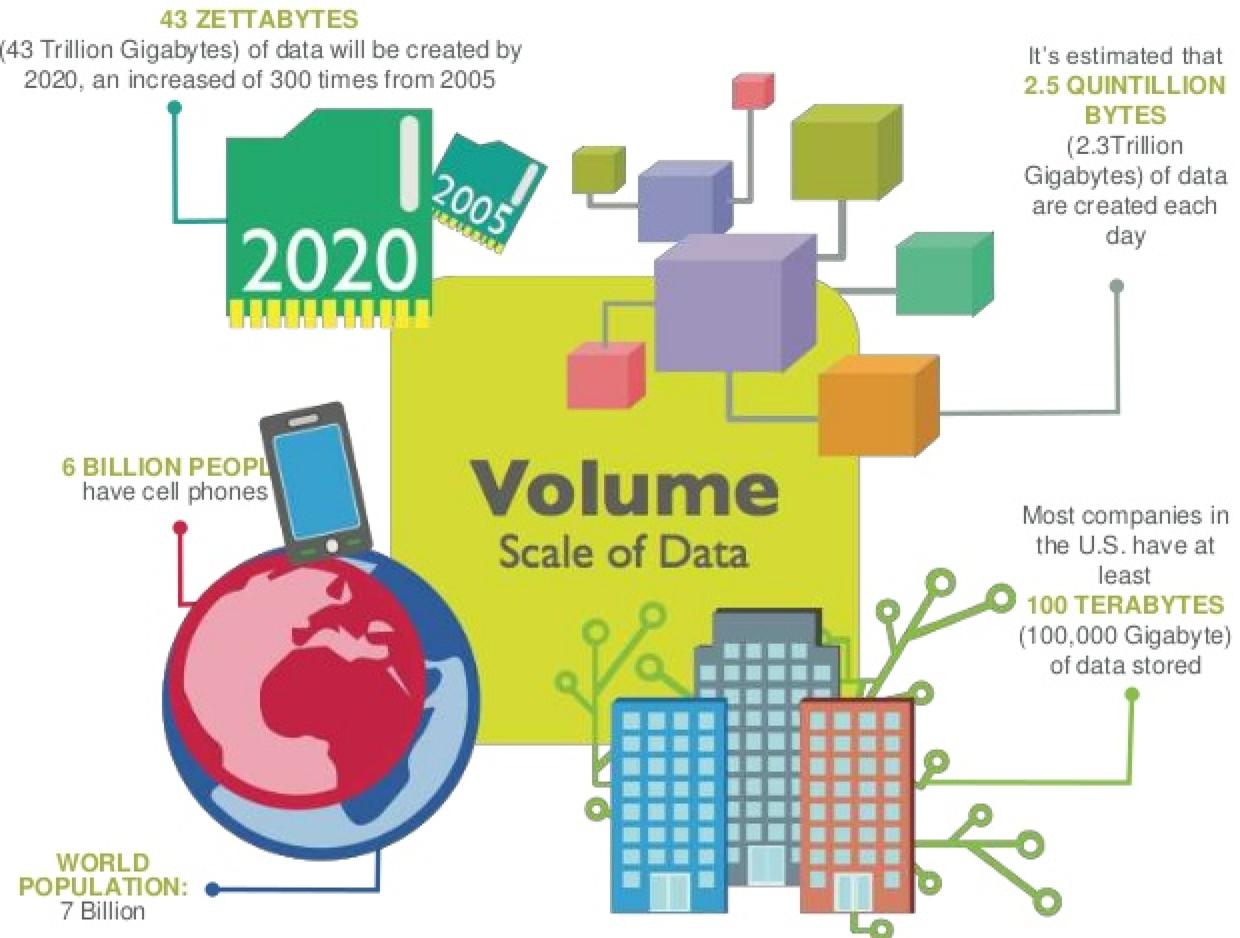
Χαρακτηριστικά Big Data

Παρακάτω, βλέπουμε αναλυτικότερα τα επτά κυριότερα χαρακτηριστικά των μεγάλων δεδομένων – 7 Vs – τα οποία αυξήθηκαν και θα συνεχίσουν να αυξάνονται από το 2001[31] που αριθμούσαν αρχικά στα 3 Vs.

1. Volume (Όγκος).

   Το μέγεθος των δεδομένων. Είναι το πιο γνωστό χαρακτηριστικό των μεγάλων δεδομένων. Σύμφωνα με νέα έρευνα της IBM[33] το 2017, κάθε μέρα δημιουργούνται 2.5 quintillion bytes δεδομένων, δηλαδή 2.5 x 10^{18} bytes ή 2.3 τρισεκατομμύρια Gigabytes. Αυτό είναι λογικό τη

στιγμή που 300 συνολικά ώρες βίντεο "ανεβαίνουν" στο YouTube κάθε λεπτό[34], ενώ το 2016 μετρήθηκε η χρήση Internet μέσω του κινητού σε παγκόσμιο επίπεδο στα 7 Exabytes κάθε μήνα[35]. Δύο από τις κυριότερες πηγές αδόμητων δεδομένων στις μέρες μας αποτελούν τα Social Media[35] και το Internet of Things[36].



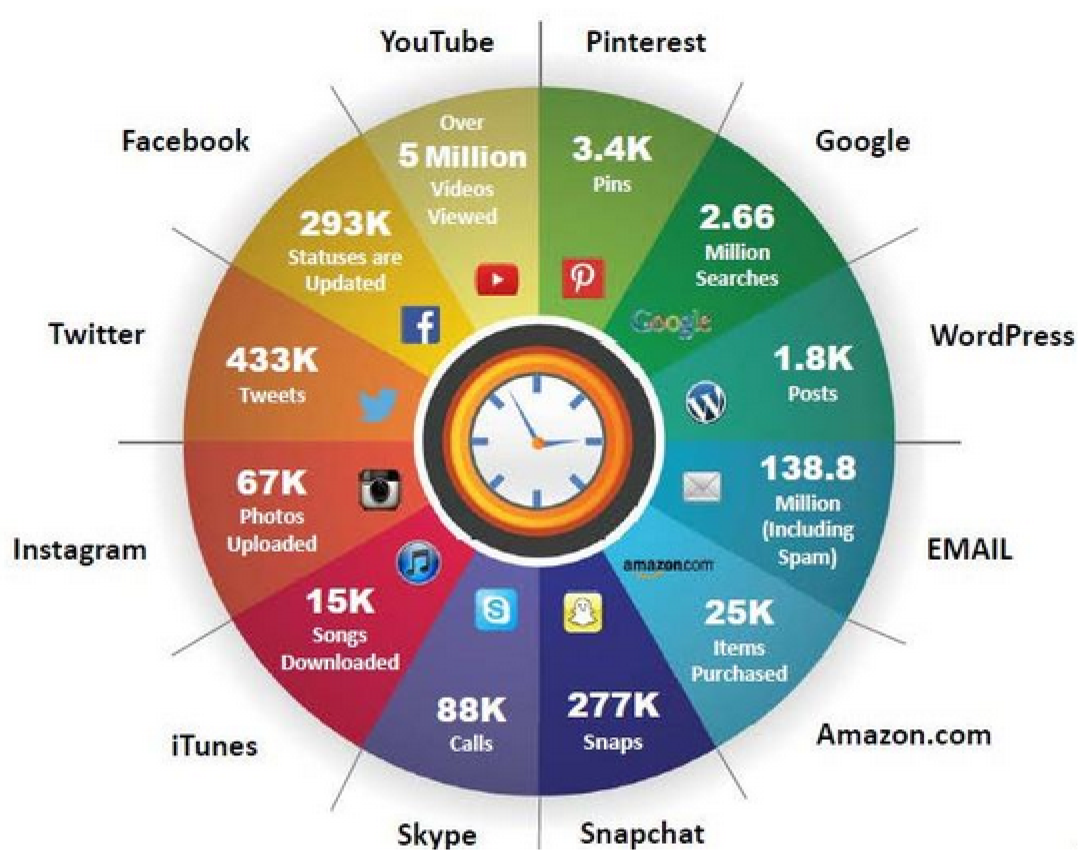
Εικόνα 2: Ο όγκος των Big Data παγκοσμίως.

2. Velocity (Ταχύτητα).

   Η ταχύτητα δημιουργίας, αναπαραγωγής, γέννησης ή ανανέωσης των δεδομένων. Αυτή η ταχύτητα αυξάνει συνεχώς τις απαιτήσεις για επεξεργασία, ανάλυση και αποθήκευση των δεδομένων. Η Google επεξεργάζεται περισσότερες από 40.000 αναζητήσεις το δευτερόλεπτο, το οποίο μεταφράζεται περίπου σε 3.5 δισεκατομμύρια αναζητήσεις κάθε μέρα[37]. Ακόμη, το Facebook ισχυρίζεται ότι λαμβάνει περίπου 600 Terabytes νέων δεδομένων κάθε μέρα[38].

3. Variety (Ποικιλία).

Τα Big Data συνήθως αναφέρονται σε δεδομένα ημι-δομημένα ή μη-δομημένα και λιγότερο σε δομημένα δεδομένα με κύρια κατηγορία τα μη-δομημένα (unstructured data). Παραδείγματα καθημερινών μεγάλων δεδομένων ποικίλης μορφής αποτελούν ήχος audio, εικόνες, βίντεο, ανανεώσεις Social Media, log αρχεία, δεδομένα σχετικά με κλικ χρηστών, δεδομένα αισθητήρων κ.ά.



Εικόνα 3: What happens online in 60 seconds.

4. Variability (Μεταβλητότητα).

Η μεταβλητότητα μπορεί να αναφέρεται σε διαφορετικά πράγματα όταν μιλάμε για Big Data. Ένα, είναι ο αριθμός των ασυνεπιών στα δεδομένα (inconsistencies), οι οποίες πρέπει πρώτα να εντοπιστούν από ειδικές μεθόδους προτού γίνει ανάλυση, επεξεργασία ή αποθήκευση. Ακόμη, μπορεί να αναφέρεται στην διαφορετικότητα των τύπων δεδομένων μεταξύ των δεδομένων που συλλέγονται, για παράδειγμα unstructured data σε NoSQL βάσεις δεδομένων. Επίσης, μπορεί να εννοείται ως μεταβλητότητα η αλλαγή της σημασίας των δεδομένων.

5. Veracity (Αξιοπιστία).

Πρόκειται για την εγκυρότητα των δεδομένων. Το πρόβλημα εδώ είναι ότι όσο αυξάνουν τα υπόλοιπα Vs των Big Data, αυτομάτως η αξιοπιστία πέφτει. Η εμπιστοσύνη της πηγής, του έγκυρου περιεγχομένου και η σημασία της ανάλυσης των δεδομένων περιγράφουν κατά κάποιον τρόπο την ιδιότητα αυτή. Για παράδειγμα, εάν μας δωθούν ως πληροφορίες οι πωλήσεις των προϊόντων μιας εταιρίας τα τελευταία πέντε έτη, τότε αυτόματα θα θέλουμε να μάθουμε συγκεκριμένα πράγματα για αυτά τα δεδομένα. Όπως, ποιοι δημιούργησαν την λίστα πωλήσεων που λάβαμε και ποια μεθοδολογία ακολούθησαν στη συλλογή δεδομένων πωλήσεων. Ή ακόμη αν τροποποιήθηκε η λίστα των πωλήσεων από τότε που μετρήθηκε. Οι απαντήσεις σε αυτές τις ερωτήσεις θα μας βοηθήσουν να κρίνουμε την αξιοπιστία της πληροφορίας που έχουμε στα χέρια μας. Όσο μεγαλύτερη είναι η αξιοπιστία σε ένα σύνολο από δεδομένα, τόσο περισσότερο κατανοούνται οι κίνδυνοι με την ανάλυση και τη λήψη επιχείρησιακών αποφάσεων με βάση αυτά τα δεδομένα.

6. Visualization (Απεικόνιση).

Η απεικόνιση ή οπτικοποίηση των δεδομένων παρουσιάζει βαθμούς δυσκολίας στην υλοποίησή της. Τα σύγχρονα εργαλεία απεικόνισης αντιμετωπίζουν τεχνικές δυσκολίες λόγω σε περιορισμούς στην τεχνολογία μνήμης RAM, στην κλιμάκωση, την λειτουργικότητα και τον χρόνο απόκρισης. Για παράδειγμα, δεν είναι εφικτό να βασιζόμαστε σε παραδοσιακούς τρόπους απεικόνισης γράφων, όταν θέλουμε να οπτικοποιήσουμε δισεκατομμύρια σημεία. Συνεπώς, χρειαζόμαστε νέους τρόπους όπως το clustering, το treemapping, τα sunbursts, οι παράλληλες συντεταγμένες, τα circular διαγράμματα δικτύων ή τεχνικές που χρησιμοποιούν cone trees. Συνδυάζοντας την πολυπλοκότητα των τελευταίων με την unstructured φύση των Big Data μπορούμε να απεικονίσουμε την πληροφορία.

7. Value (Αξία).

Το σημαντικότερο από τα χαρακτηριστικά των μεγάλων δεδομένων είναι η ίδια η αξία τους. Ειδικά για μία επιχείρηση, η επεξεργασία και ανάλυση των Big Data αυτής σε ετήσια βάση, είναι απαραίτητη για τη σωστή λήψη αποφάσεων και τον καθορισμό της επόμενης στρατηγικής κίνησης. Ο έλεγχος των υπόλοιπων χαρακτηριστικών των μεγάλων δεδομένων αποσκοπούν κάπου, μόνο όταν γεννιέται αξία και συμπεράσματα χρήσιμα μέσα από τα δεδομένα. Ουσιαστική αξία, για παράδειγμα, είναι η κατανόηση των αναγκών των πελατών και η κατάλληλη προσέγγιση, ή η βελτιστοποίηση των διεργασιών της εταιρίας βοηθώντας στην απόδοση κερδών.

Στη συνέχεια, βλέπουμε κάποιες σύγχρονες ευκαιρίες και εφαρμογές των Big Data σε παγκόσμιο επίπεδο και με ποιον τρόπο μπορούν να επηρεάσουν τον τρόπο ζωής και την σύγχρονη κοινωνία.

Ευκαιρίες των Big Data

Το μέγεθος των δεδομένων αυξάνεται παγκοσμίως και ταυτόχρονα οι αποθηκευτικές μας συσκευές γίνονται μικρότερες σε μέγεθος και πιο ισχυρές σε επιδόσεις. Με αυτήν την αφθονία δεδομένων, γίνεται όλο και μεγαλύτερη εστίαση στους τρόπους διαχείρισης, εξόρυξης και χρήσης των δεδομένων που αποθηκεύουμε. Πρόσφατες έρευνες εκτιμούν ότι γίνεται παγκόσμια επεξεργασία πάνω από 295 Exabytes δεδομένων – όπως είδαμε 2.3 Exabytes καθημερινά.

Η Google κατέχει την μεγαλύτερη χωρητικότητα αποθήκευσης δεδομένων από οποιαδήποτε άλλη εταιρία, ενώ ταυτόχρονα, εταιρίες, όπως National Security Agency (NSA), Facebook, Microsoft, Amazon, Chevron κ.ά. που προσπαθούν να τη συναγωνιστούν, κτίζουν και αυτές τεράστια data centers σε όλα τα μέρη του κόσμου, όπως διατηρεί η Google.

Το Facebook συλλέγει καθημερινά έως και 600 Terabytes δεδομένων, ενώ τα αποθηκευμένα βίντεο και φωτογραφίες σε Big Data τεχνολογίες όπως Hive, HDFS κ.ά. αριθμούν πάνω από 100 Petabytes δεδομένων[38]. Τα περισσότερα δεδομένα είναι σχετικά με δραστηριότητες χρηστών και ανανεώσεις status. Με περισσότερους από 1 δισεκατομμύριο ενεργούς χρήστες, το Facebook διαχειρίζεται περίπου 2.5 δισεκατομμύρια αντικείμενα, όπως σχόλια, δημοσιεύσεις κλπ., ενώ αυτοί οι χρήστες "ανεβάζουν" πάνω από 300 εκατομμύρια φωτογραφίες καθημερινά. Με την επιτυχή επεξεργασία και ανάλυση όλων αυτών των πληροφοριών για τον κάθε χρήστη, το Facebook μπορεί και προτείνει τις κατάλληλες διαφημίσεις, πράγμα που δείχνει ξεκάθαρα τη σωστή χρήση του Value χαρακτηριστικού των μεγάλων δεδομένων. Παρακάτω, αναλύουμε το Cloud Computing και τις πτυχές του στην σύγχρονη κοινωνία.

Cloud Computing

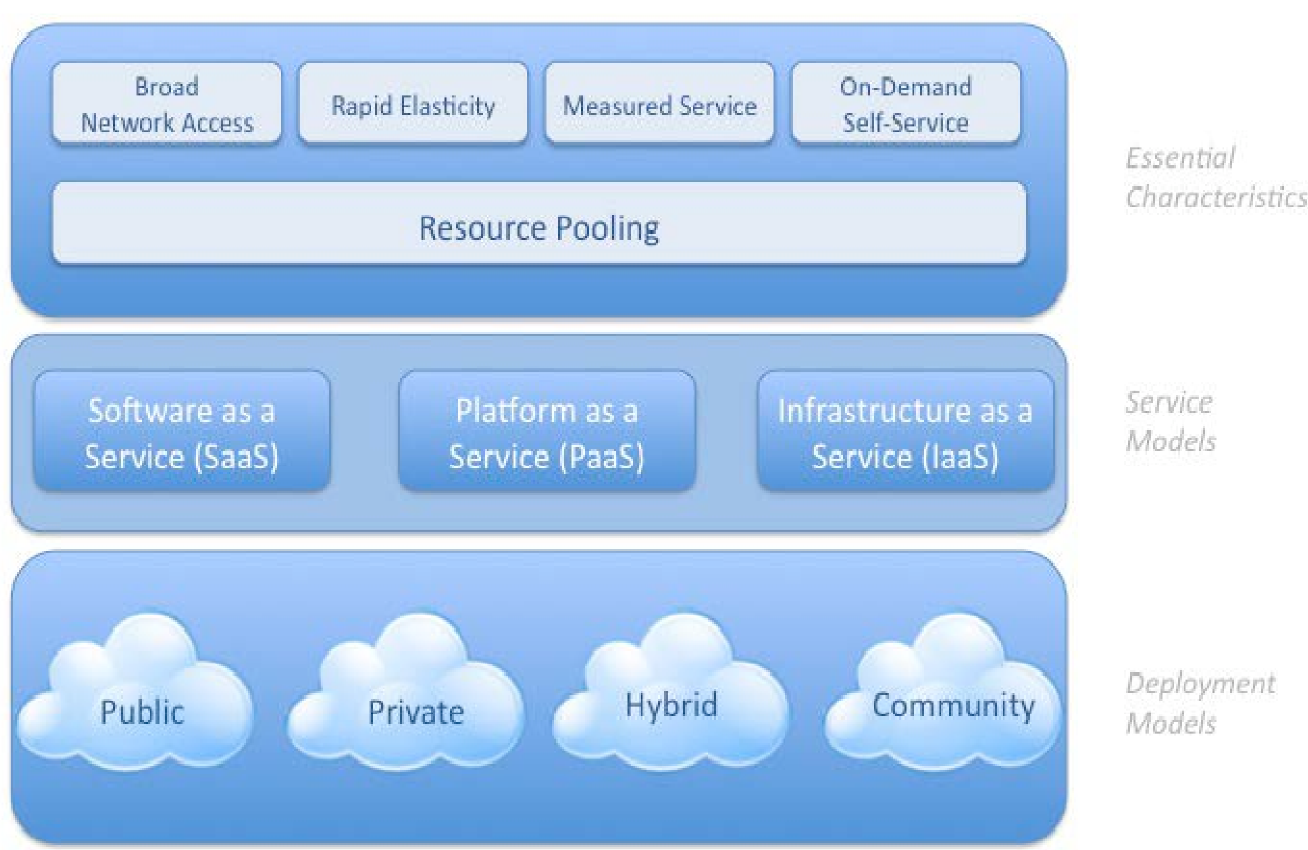
Εισαγωγή

Το Cloud, μαζί με τα χαρακτηριστικά και τα μοντέλα που προσφέρει, αποτελεί αδιαμφισβήτητα μία από τις πιο εύχρηστες, χρήσιμες και οικονομικές τεχνολογίες της Επιστήμης των Υπολογιστών σήμερα. Η ανάγκη εταιριών, όπως Google, Amazon, Facebook, IBM κτλ., για ανταπόκριση σε

αυξημένες απαιτήσεις αποθηκευτικού χώρου και υπολογιστικής ισχύος (αίτιο, η ύπαρξη Big Data), παράλληλα με την χρήση πληθώρας συσκευών συνδεδεμένων στο Internet από τελικούς χρήστες, συνέβαλαν στην ανάπτυξη της τεχνολογίας του Υπολογιστικού Νέφους (Cloud Computing).

Ορισμός

Από όλους τους ορισμούς και τα παραδείγματα για το Cloud, αξιοσημείωτος είναι ο επίσημος ορισμός του NIST[1] (National Institute of Standards and Technology). Σύμφωνα με το Εθνικό Ινστιτούτο Προτύπων και Τεχνολογίας των Ηνωμένων Πολιτειών (NIST), το Cloud computing ορίζεται ως ένα μοντέλο που παρέχει πανταχού παρούσα, ευνοϊκή και κατ' απαίτηση δικτυακή πρόσβαση σε έναν κοινόχρηστο χώρο προγραμματίσιμων υπολογιστικών πόρων (για παράδειγμα δίκτυα, εξυπηρετητές, αποθηκευτικός χώρος, εφαρμογές και υπηρεσίες) που μπορούν να διατεθούν γρήγορα με ελάχιστη προσπάθεια διαχείρισης ή αλληλεπίδρασης του πελάτη με τον πάροχο αυτών. Το μοντέλο αυτό αποτελείται από πέντε (5) Χαρακτηριστικά (Essential Characteristics), τρία (3) Μοντέλα Υπηρεσιών (Service Models) και τέσσερα (4) Μοντέλα Ανάπτυξης (Deployment Models).



Εικόνα 4: The NIST definition of Cloud computing (Πηγή: [2])

Essential Characteristics

Ο ορισμός του Cloud από το National Institute of Standards and Technology (NIST) συνεχίζει με τα 5 "Essential Characteristics" (Χαρακτηριστικά):

On-demand self-service: Ο κάθε καταναλωτής επιλέγει και προμηθεύεται τους υπολογιστικούς πόρους όπως αυτός επιθυμεί, χωρίς την αλληλεπίδραση με τον αυτό πάροχο.

Broad network access: Οι υπολογιστικοί πόροι είναι διαθέσιμοι μέσω δικτύου (από Internet μέχρι και LAN) και προσπελάσσονται με τρόπο τυποποιημένο και ανεξάρτητο από την ετερογένεια των συσκευών που μπορεί να χρησιμοποιεί ο τελικός χρήστης (μπορεί να είναι κινητό, ταμπλέτα, φορητός υπολογιστής, πλατφόρμες ή διαφορετικά λειτουργικά συστήματα).

Resource pooling: Οι υπολογιστικοί πόροι ενός παρόχου βρίσκονται σε έναν κοινόχρηστο χώρο για να μπορούν να εξυπηρετούνται αιτήματα πολλών διαφορετικών χρηστών-πελατών. Οι φυσικοί και εικονικοί πόροι διατίθενται στους πελάτες ανάλογα με τη ζήτηση. Τελικά, ο πελάτης δεν έχει αίσθηση της ακριβούς τοποθεσίας των πόρων που του διατέθηκαν (για παράδειγμα σε ποιούς επεξεργαστές τρέχει η εφαρμογή του) ή αρχείων που ενδεχομένως 'ανέβασε' στο Cloud (σε ποιον φυσικό σκληρό δίσκο), παραμόνο πιο αφηρημένα, όπως ποιο datacenter, ποια πόλη ή ποια χώρα - ανάλογα με τον πάροχο των πόρων. Παραδείγματα υπολογιστικών πόρων αποτελούν ο αποθηκευτικός χώρος, οι επεξεργαστές, η μνήμη, το εύρος ζώνης δικτύου.

Rapid elasticity: Οι υπολογιστικοί πόροι διατίθενται ελαστικά, κάποιες φορές αυτόματα, ώστε να δεσμεύονται και να αποδεσμεύονται γρήγορα ανάλογα με τη ζήτηση.

Measured service: Τα Cloud computing συστήματα προσαρμόζουν την χρέωση των πελατών τους ανάλογα με το είδος των πόρων που προφέρουν (είτε είναι αποθηκευτικός χώρος, είτε υπολογιστική ισχύς - πολλών - επεξεργαστών, εύρος ζώνης δικτύου, αριθμός user account που επιτρέπονται στον πελάτη, κ.ά.) και τη χρησιμοποίηση αυτών (για παράδειγμα pay-per-use ή charge-per-use χρέωση). Η χρησιμοποίηση υπολογιστικών πόρων μετράται και αναφέρεται με διαφάνεια ανάμεσα σε πάροχο και πελάτη.

Από την Εικόνα 4, ορίστηκαν τα πέντε (5) "Essential Characteristics" του Cloud σύμφωνα με το NIST, τα οποία, όπως είδαμε, αποτελούν τα στοιχειώδη χαρακτηριστικά που δεν λείπουν από κάθε Cloud computing σύστημα. Στη συνέχεια, θα αναφερθεί σύντομα το κομμάτι του ορισμού για τα "Service Models" (Μοντέλα Υπηρεσιών). Εδώ το Cloud χωρίζεται ανάλογα με το είδος της υπηρεσίας που προσφέρει στον εκάστοτε πελάτη.

Σε αυτό το σημείο επιβάλεται να αναφερθεί ότι μια Υποδομή Νέφους (Cloud Infrastructure) ορίζεται ως η συλλογή εκείνη υλικού και λογισμικού που σαν σύνολο περιέχει τα πέντε (5) Χαρακτηριστικά του Cloud computing ("Essential characteristics"). Αυτή η συλλογή χωρίζεται σε physical και

abstraction layers, με την πρώτη να αποτελείται από τους απαραίτητους πόρους υλικού (όπως εξυπηρετητές, αποθηκευτικό χώρο και μηχανήματα δικτύου) που θα υποστηρίξουν παρεχόμενες υπηρεσίες Cloud, και την δεύτερη (abstraction layer) από το κατάλληλο λογισμικό που θα αναπτυχθεί πάνω στην psysical layer, αναδεικνύοντας τα πέντε (5) "Essential Characteristics" του Cloud computing.

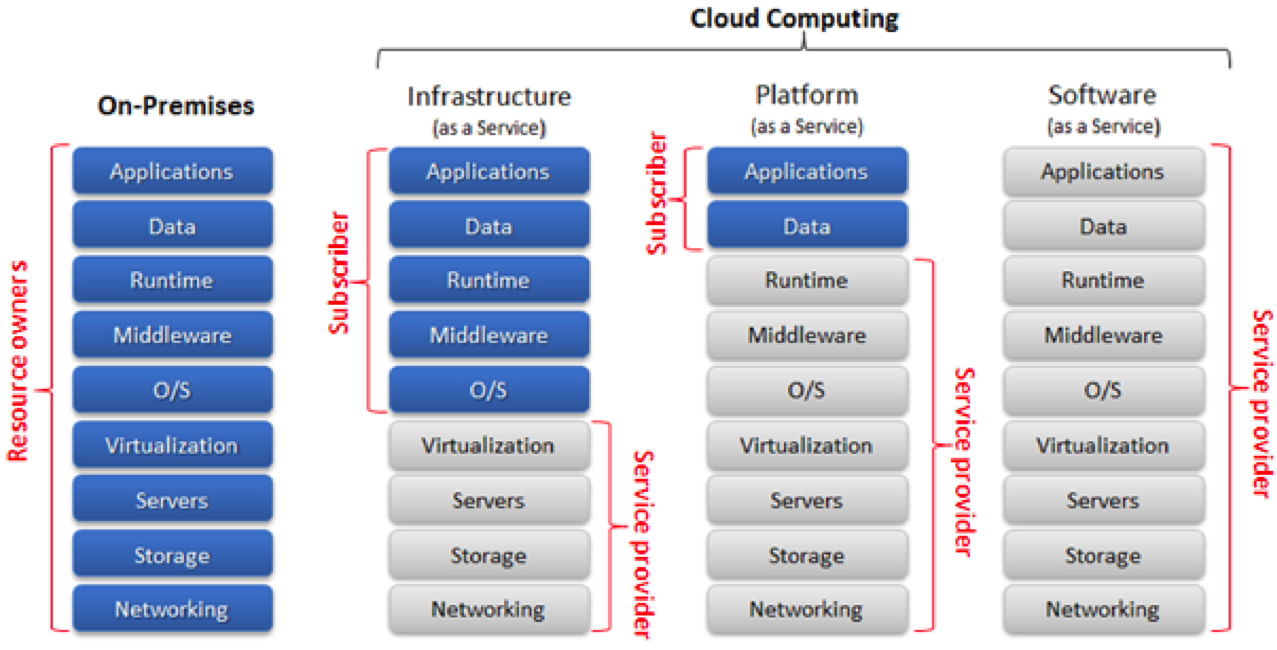
Service Models

Η ετερογένεια των υπηρεσιών που μπορεί να προσφέρει το Cloud ποικίλει και για αυτό το λόγο το NIST αναφέρει τα τρία (3) Μοντέλα Υπηρεσιών που μαζί αποτελούν το SPI Model:

SaaS - Software as a Service: Σε αυτό το μοντέλο υπηρεσίας, ο πελάτης μπορεί να χρησιμοποιεί τις εφαρμογές που διαθέτει ο πάροχος SaaS σε μια Υποδομή Νέφους. Η πρόσβαση του πελάτη στην εφαρμογή γίνεται μέσα από έναν φυλλομετρητή ή ένα πρόγραμμα-πελάτη. Εδώ, ο πελάτης δεν διαχειρίζεται και δεν ελέγχει την Υποδομή Νέφους όπου αναπτύσσονται οι εφαρμογές που χρησιμοποιεί (Cloud Infrastructure - συμπεριλαμβανομένων εικονικού δικτύου, εξυπηρετητών, λειτουργικών συστημάτων, αποθηκευτικού χώρου ή επιπλέον δυνατοτήτων μιας συγκεκριμένης εφαρμογής), με πιθανή περίπτωση εξαίρεσης περιορισμένες ρυθμίσεις συγκεκριμένης εφαρμογής. Παραδείγματα παρόχων SaaS: Oracle, SAP, Cobweb, MuleSoft, Salesforce, Akamai, CA Technologies, Adobe, Microsoft, Intuit κ.ά.

PaaS - Platform as a Service: Σε αυτό το μοντέλο υπηρεσίας, ο πελάτης μπορεί να αναπτύσσει και να διαχειρίζεται στην Υποδομή Νέφους (Cloud Infrastructure) δικές του εφαρμογές, χρησιμοποιώντας τα εργαλεία, τις γλώσσες προγραμματισμού και τις βιβλιοθήκες που είναι συμβατές και υποστηρίζει ο πάροχος PaaS. Εδώ, ο πελάτης δεν διαχειρίζεται και δεν ελέγχει την Υποδομή Νέφους (Cloud Infrastructure - συμπεριλαμβανομένων εικονικού δικτύου, εξυπηρετητών, λειτουργικών συστημάτων, αποθηκευτικού χώρου), αλλά έχει τον έλεγχο των εφαρμογών που αναπτύσσονται, πιθανώς και των ρυθμίσεων του περιβάλλοντος που τις φιλοξενεί. Παραδείγματα παρόχων PaaS: Salesforce App Cloud: Heroku Enterprise, AWS EC2, Microsoft Azure, Salesforce App Cloud: Force.com, OpenShift, Zoho Creator, Morpheus, Dokku, Google App Engine, AWS Elastic Beanstalk, AWS Lambda, IBM Bluemix, Cloud Foundry, Deis κ.ά.

IaaS - Infrastructure as a Service: Εδώ ο πελάτης μπορεί να χρησιμοποιήσει εικονικά επεξεργαστές, αποθηκευτικό χώρο, λειτουργικά συστήματα, δίκτυο και άλλους υπολογιστικούς πόρους που διατίθενται από τον πάροχο IaaS. Ο πελάτης δεν διαχειρίζεται και δεν ελέγχει την Υποδομή Νέφους (Cloud Infrastructure, psysical layer - βλ. παραπάνω), αλλά μπορεί να ελέγξει τους εικονικούς υπολογιστικούς πόρους που του διατίθενται (επεξεργαστές, αποθηκευτικός χώρος, λειτουργικά συστήματα, δίκτυο). Παραδείγματα παρόχων IaaS: Amazon AWS, Windows Azure, Google Compute Engine, Rackspace Open Cloud, IBM SmartCloud Enterprise, HP Enterprise Converged Infrastructure κ.ά.



Εικόνα 5: Τα 3 Μοντέλα Υπηρεσιών ("Service Models") του Cloud computing. (Πηγή: [3])

Η λογική της ετερογένειας των υπηρεσιών που προσφέρει το Cloud σήμερα, λέγεται "**Anything As A Service**" ή "**Everything As A Service**" και εδώ βλέπουμε κάποια από τα μοντέλα υπηρεσιών της αγοράς:

- Storage as a Service (SaaS), για παράδειγμα NetApp SaaS Solutions
- Database as a Service (DaaS), για παράδειγμα MongoDB Atlas DaaS
- Container as a Service (CaaS), για παράδειγμα Docker CaaS
- Malware as a Service (MaaS), μιλώντας πλέον για Cloud στο Darknet με παράνομους πωλητές που παρέχουν σε πελάτες υπηρεσία ransomware ή malware ή πρόσβαση σε computer botnets κακόβουλου λογισμικού, τα οποία χρησιμοποιούνται ως υπηρεσία επίθεσης σε κάποιον στόχο.
- Network as a Service (NaaS), για παράδειγμα Aryaka NaaS Solutions
- Monitoring as a Service (MaaS), για παράδειγμα Hewlett Packard Enterprise MaaS
- Windows as a Service (WaaS), για παράδειγμα Microsoft WaaS
- Desktop as a service (DaaS), για παράδειγμα Amazon WorkSpaces DaaS
- Disaster Recovery as a Service (DRaaS), για παράδειγμα IBM DRaaS

Παρόλα αυτά, το SPI Model που παρουσιάστηκε προηγουμένως (SaaS/PaaS/IaaS) υπερκαλύπτει όλα τα πιθανά σενάρια μοντέλων υπηρεσίας που ορθά αντικαθιστούν το "Anything" ή "Everything", βλ. Εικόνα 5.

Deployment Models

Το NIST ορίζει τα 4 Μοντέλα Ανάπτυξης ("Deployment Models"), μοντέλα που αφορούν περισσότερο την ασφάλεια και την ιδιωτικότητα στο Cloud:

Private Cloud: Παρέχεται για αποκλειστική χρήση από ένα μοναδικό οργανισμό που περιλαμβάνει πολλαπλούς χρήστες (για παράδειγμα χρήστες στα διαφορετικά γραφεία μιας επιχείρησης). Εδώ, το παρεχόμενο Cloud μπορεί να το κατέχει, να το διαχειρίζεται και να το λειτουργεί ο ίδιος ο οργανισμός, ή τρίτος ή κάποιος συνδυασμός αυτών και μπορεί να υπάρχει εντός ή εκτός των εγκαταστάσεων του οργανισμού.

Community Cloud: Παρέχεται για αποκλειστική χρήση από μια συγκεκριμένη κοινότητα πελατών από οργανισμούς που έχουν κοινούς στόχους (για παράδειγμα αποστολή, απαιτήσεις ασφάλειας, πολιτική, εκτιμήσεις συμμόρφωσης). Εδώ, το παρεχόμενο Cloud μπορεί να το κατέχει, να το διαχειρίζεται και να το λειτουργεί ένας ή περισσότεροι από τους οργανισμούς της κοινότητας, ή τρίτος, ή κάποιος συνδυασμός αυτών και μπορεί να υπάρχει εντός ή εκτός των εγκαταστάσεων του οργανισμού.

Public Cloud: Παρέχεται για ανοιχτή χρήση από το οποιονδήποτε και μπορεί να το κατέχει, να το διαχειρίζεται και να το λειτουργεί ένας επιχειρηματικός ή ακαδημαϊκός ή κυβερνητικός οργανισμός ή ένας συνδυασμός αυτών. Η φυσική του τοποθεσία είναι στις εγκαταστάσεις του πάροχου του Cloud.

Hybrid Cloud: Αποτελεί έναν συνδυασμό δύο ή περισσότερων των παραπάνω διακριτών περιπτώσεων (Private, Community ή Public υποδομή), που παραμένουν μοναδικές οντότητες αλλά συνδέονται μεταξύ τους από τεχνολογία (πρότυπη ή ιδιόκτητη) η οποία επιτρέπει τη φορητότητα των δεδομένων (Big Data) και των εφαρμογών (για παράδειγμα εξισορρόπηση φορτίου μεταξύ των υποδομών Cloud που συμμετέχουν).

2

Τα Big Data, όπως είδαμε προηγουμένως, αποτελούν κύριο σημείο μελέτης της επιστήμης των υπολογιστών στις μέρες μας. Τα χαρακτηριστικά τους είναι αυτά που ορίζουν τις νέες προκλήσεις που καλείται να αντιμετωπίσει κάθε επιχείρηση ή μηχανικός. Σε αυτό το κεφάλαιο αναλύουμε και περιγράφουμε τα Big Data υπό το πρίσμα τρόπων επεξεργασίας και βελτιστοποίησής τους.

Βάσεις Δεδομένων

Εισαγωγή

Τα συστήματα διαχείρισης βάσεων δεδομένων κρίνονται απαραίτητα για τη διαχείριση μεγάλου όγκου δεδομένων (big data) στη σύγχρονη εποχή. Οποιαδήποτε ηλεκτρονική εφαρμογή, και κυρίως διαδικτυακή, τις περισσότερες φορές αλληλεπιδρά με μια βάση δεδομένων. Οι αγορές προϊόντων μέσω Internet και οι τραπεζικές συναλλαγές είναι μερικά από τα πιο συχνά παραδείγματα χρήσης βάσεων δεδομένων. Ας ορίσουμε τώρα καλύτερα αυτήν την έννοια.

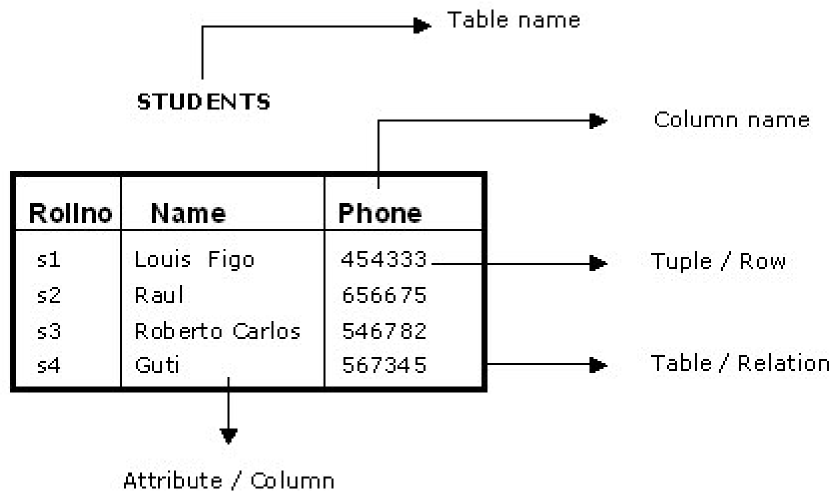
Βάση δεδομένων είναι μια σειρά από δεδομένα που σχετίζονται μεταξύ τους. Στην επιστήμη των υπολογιστών αυτά τα δεδομένα βρίσκονται σε έναν ή περισσότερους υπολογιστές και προσπελάσσονται με έναν ή περισσότερους διαφορετικούς τρόπους. Με τον όρο δεδομένα εννοούμε πληροφορίες για κάποιο συμβάν που έχουν αποθηκευτεί. Κατά το σχεδιασμό και την κατασκευή μιας βάσης δεδομένων, η οργάνωσή της και οι τρόποι προσπέλασης της βασίζονται σε πραγματικά σενάρια αναζήτησης και επεξεργασίας δεδομένων της καθημερινής ζωής. Για παράδειγμα, η εύρεση και κράτηση μιας αεροπορικής θέσης βολεύει να οργανωθεί έτσι ώστε να φαίνεται πιο καθαρά ποιες μέρες υπάρχουν ακόμα κενές θέσεις και ποιες όχι. Οι σχεδιαστές βάσεων δεδομένων λαμβάνουν πάντα υπόψιν τους τις ανάγκες και απαιτήσεις της σύγχρονης εποχής.

Ακόμη ένα σημαντικό σημείο, σχετικό με τις βάσεις δεδομένων, είναι τα εργαλεία διαχείρισης και συντήρησης αυτών, τα οποία χρησιμοποιούνται κατά κόρον. Ένα τέτοιο εργαλείο ονομάζεται Σύστημα Διαχείρισης Βάσεων Δεδομένων (Database Management System – DBMS) και δίνει τη δυνατότητα στους χρήστες του να ορίζουν, να δημιουργούν, να θέτουν ερωτήματα, να συντηρούν

και να διαχειρίζονται βάσεις δεδομένων. Άλλες χρήσιμες λειτουργίες που προσφέρει είναι ασφάλεια, αποθήκευση αντιγράφων (backup) και παρακολούθηση στατιστικών και άλλων μετρήσεων στη βάση (monitoring). Τα διαφορετικά μοντέλα DMBS που εμφανίζονται και χρησιμοποιούνται μέχρι και σήμερα είναι το σχεσιακό μοντέλο (Relational – RDBMS), το αντικειμενοστραφές-σχεσιακό (Object-Relational – ORDBMS), το αντικειμενοστραφές (Object – ODMBS) και το NoSQL μοντέλο, το οποίο θεωρείται το πλέον μέτα-σχεσιακό μοντέλο (post-relational) και εισήγαγε νέες έννοιες και λογικές στο χώρο. Τα δύο πιο διαδεδομένα μοντέλα είναι το RDBMS και το NoSQL και θα τα δούμε στη συνέχεια.

Σχεσιακό Σύστημα Διαχείρισης Βάσεις Δεδομένων – RDBMS

Το σχεσιακό μοντέλο είναι από τα πιο βασικά και διαδεδομένα μοντέλα και χρησιμοποιείται μέχρι σήμερα σε πολλά και δημοφιλή συστήματα διαχείρισης βάσεων δεδομένων. Εμφανίστηκε για πρώτη φορά από τον E. F. Codd στα τέλη της δεκαετίας του '70[15], εισάγοντας ένα νέο τότε τρόπο διαχείρισης της πληροφορίας, όπου όλα τα δεδομένα τοποθετούνται σε πλειάδες και οργανώνονται σε σχέσεις μεταξύ τους. Η προσέγγιση του Codd χρησιμοποιεί κατηγορηματικό λογισμό πρώτης τάξης και δίνει την ευκαιρία στον χρήστη να απαιτεί το είδος της πληροφορίας που θα περιέχει η βάση και να ζητάει άμεσα και ξεκάθαρα την πληροφορία που χρειάζεται, αφήνοντας έτσι το λογισμικό του DBMS να περιγράψει την δομή αποθήκευσης των δεδομένων και να βρει τα δεδομένα για να απαντήσει στα ερωτήματα. Παρακάτω, φαίνεται με ένα σύντομο παράδειγμα ένας πίνακας σχεσιακού μοντέλου εν ονόματι "STUDENTS":



Εικόνα 6: Relational DBMS

Σε αυτές τις βάσεις τηρείται ένα σύνολο κανόνων[16] που εξασφαλίζει μοναδικότητα στις εγγραφές (δεν πρέπει να υπάρχουν ίδιες εγγραφές στους πίνακες) και ακεραιότητα οντότητας και σχέσης (entity and referential integrity). Όταν λείπει ένα πεδίο μιας εγγραφής, τότε το DBMS συμπληρώνει με null. Τη σημασία του τελευταίου κανόνα θα δούμε αργότερα συκρίνοντας με τις NoSQL βάσεις δεδομένων και ειδικά για μεγάλο όγκο αδόμητων δεδομένων.

Όπως είδαμε παραπάνω το σύστημα διαχείρισης βάσεων δεδομένων για σχεσιακές βάσεις λέγεται RDBMS. Τα πιο γνωστά και διαδεδομένα RDBMS είναι τα Oracle Database και MySQL της Oracle, Microsoft SQL Server, DB2 και Informix της IBM, Sybase Adaptive Server Enterprise, Sybase IQ και Teradata. Σε αυτά τα συστήματα γίνεται εκτενής χρήση της γλώσσας SQL (Structured Query Language)[17]. Η SQL είναι εξ ορισμού και κατασκευής κατάλληλη για Relational DMBS συστήματα. Εισήγαγε τη δυνατότητα πρόσβασης σε μια ή περισσότερες εγγραφές με μία εντολή και εξάλειψε την ανάγκη να προσδιορίζεται ο τρόπος πρόσβασης σε μια εγγραφή, λ.χ. με ή χωρίς δείκτη. Η SQL μπορεί να δημιουργήσει (CREATE) και να καταστρέψει (DROP) στιγμιότυπα βάσεων, αλλά και να εκτελεί τις ακόλουθες ενέργειες επί της αποθηκευμένης πληροφορίας INSERT, SELECT, UPDATE, DELETE, JOIN.

NoSQL Βάσεις Δεδομένων

Η ονομασία NoSQL (non SQL) παραπέμπει στη νέα γενιά μετά-σχεσιακών (next generation post-relational databases) βάσεων δεδομένων όπως είναι οι MongoDB, Cassandra, Counchbase, Virtuoso, ArangoDB, κλπ. Πολλές φορές ο ορισμός μεταφράζεται ως "Not only SQL" τονίζοντας ότι υποστηρίζουν και SQL-τύπου ερωτήματα. Αυτές οι βάσεις είχα ήδη κάνει την εμφάνισή τους από τα τέλη της δεκαετίας του '60, χωρίς τη σχετική ονομασία μέχρι τις αρχές του 21ου αιώνα, όπου έγιναν πολύ δημοφιλείς. Κι αυτό γιατί άρχισαν να χρησιμοποιούνται όλο και περισσότερο σε επεξεργασία και εξορύξεις από μεγάλο όγκο δεδομένων (big data) και real-time εφαρμογές και κυρίως από μεγαθήρια εταιρίες στο χώρο, όπως Google, Facebook, Amazon. Η απλούστερη σχεδίασή τους από τις σχεσιακές βάσεις και η ευκολία οριζόντιας κλιμάκωσης σε συμπλέγματα (horizontal scaling to clusters – κάτι που αδυνατούν να κάνουν οι σχεσιακές εύκολα και γρήγορα) μαζί με τον καλύτερο έλεγχο διαθεσιμότητας (availability) είναι τα κύρια σημεία που ανάγκασαν τις μεγάλες εταιρίες να στραφούν στην υιοθέτησή τους. Στη συνέχεια, εξηγούμε τον βασικό λόγο που μπορούν οι NoSQL βάσεις να προσφέρουν όλα τα παραπάνω, ενώ οι σχεσιακές θεωρούνται ακατάλληλες.

Ο λόγος αυτός είναι το δυναμικό σχήμα (dynamic schema) που υποστηρίζουν οι NoSQL βάσεις δεδομένων, πράγμα άγνωστο στις σχεσιακές. Οι τελευταίες λειτουργούν με την προϋπόθεση ότι πριν την εισαγωγή δεδομένων, γνωρίζουν το σχήμα αυτών, δηλαδή τα πεδία, τους πίνακες, τα κλειδιά, κλπ. Το μειονέκτημα εδώ είναι ότι στα σύγχρονα περιβάλλοντα των μεγάλων δεδομένων (big data) τις περισσότερες φορές έχουμε να αντιμετωπίσουμε τεράστιο όγκο δεδομένων που δεν γνωρίζουμε τη μορφή και το σχήμα τους. Σε πολλές περιπτώσεις, μετά από σύντομο χρονικό διάστημα, χρειάζεται τροποποίηση κάποιου πεδίου, πράγμα που συμβαίνει συνέχεια στα περιβάλλοντα που διαχειρίζονται big data. Έτσι, θα πρέπει να αδειάσουμε τις οποιεσδήποτε σχεσιακές βάσεις, να ορίσουμε το νέο σχήμα και να επανεισάγουμε τα δεδομένα, πράγμα άκρως χρονοβόρο και ακριβό για μια υπηρεσία ή προϊόν που πρέπει να βρίσκεται online συνεχώς.

Το πρόβλημα αυτό έρχονται να λύσουν οι NoSQL βάσεις δεδομένων υποστηρίζοντας δυναμικό σχήμα. Τώρα μπορεί και ο διαχειριστής και ο χρήστης της βάσης να κάνει τέτοιες αλλαγές σε πραγματικό χρόνο με απλές εντολές, χωρίς να διακόπτεται η λειτουργία οποιωνδήποτε συστημάτων μπορεί να εξυπηρετεί η βάση. Το δυναμικό σχήμα συμβάλλει και στην απλοποίηση εντολών εισαγωγής, εξόρυξης δεδομένων και διαχειριστικών ενεργειών. Στη συνέχεια αναφέρουμε τα πιο σημαντικά είδη NoSQL βάσεων.

Είδη NoSQL βάσεων δεδομένων

Τα σημαντικότερα είδη NoSQL βάσεων αναφέρονται παρακάτω:

Εγγραφο-κεντρικές βάσεις δεδομένων (βασιζόμενες σε έγγραφα – Document-based databases): Αυτές δημιουργούν ζεύγη κλειδιών με πολύπλοκες δομές, γνωστές ως έγγραφα (document). Παρόλο που κάθε υλοποίηση document-based βάσης, διαφέρει στις λεπτομέρειες του ορισμού της με άλλες, όλες υποθέτουν ότι ένα "document" διαθέτει την πληροφορία σε πρότυπες κωδικοποιήσεις, όπως XML, YAML, JSON, αλλά και BSON (Binary JSON). Τα έγγραφα αντιστοιχίζονται μέσα στη βάση από ένα μοναδικό κλειδί (key-document logic). Μία εγγραφο-κεντρική βάση (document-based) διαθέτει, ακόμη, ένα API (Application Programming Interface) ή μια γλώσσα ερωτημάτων που επιτρέπουν την εξόρυξη εγγράφων ανάλογα με το περιεχόμενό τους. Διαφορετικές document-based βάσεις οργανώνουν τα έγγραφά τους με διαφορετικό τρόπο. Πιθανοί τρόποι είναι "Collections", "Tags", "Non-visible metadata" ή "Directory hierarchies". Συγκρίνοντας με τις σχεσιακές, θα μπορούσε κανείς να αντιστοιχίσει τα "collections" με τα "tables" και τα "documents" με τις "records", αλλά διαφέρουν σημαντικά. Κάθε "record" (εγγραφή) σε έναν "table" (πίνακα) έχει την ίδια σειρά πεδίων, ενώ τα "documents" (έγγραφα) μιας "collection" (συλλογής) μπορεί να έχουν εντελώς διαφορετικά πεδία.

Key-value βάσεις δεδομένων (βασιζόμενες στο ζεύγος κλειδί-τιμή): Αυτές αναπαριστούν τα δεδομένα ως ζεύγη key-value όπου το κάθε πιθανό κλειδί (key) εμφανίζεται το πολύ μία φορά. Το "Key-Value" Μοντέλο είναι το πιο απλό μοντέλο αναπαράστασης δεδομένων και πιο εξελιγμένα μοντέλα βασίζονται σε αυτό. Για παράδειγμα, μπορεί να επεκταθεί σε ένα μοντέλο που θα διατηρεί τα κλειδιά σε λεξικογραφική σειρά με δυνανότητα εξόρυξης συγκεκριμένου εύρους κλειδιών.

Βάσεις δεδομένων Γράφου (Graph Databases): Αυτές είναι σχεδιασμένες για δεδομένα που οι μεταξύ τους σχέση αντιπροσωπεύεται καλύτερα με έναν γράφο με πεπερασμένο αριθμό σχέσεων ανάμεσα στους κόμβους. Έτσι, χρησιμοποιούνται περισσότερο για να αποθηκεύσουν πληροφορίες για δίκτυα οποιασδήποτε μορφής, όπως για παράδειγμα, κοινωνικές επαφές, δημόσια συγκοινωνία, οδικοί χάρτες.

Στηλο-κεντρικές βάσεις δεδομένων (Column-oriented databases): Αυτές έχουν πολλές ομοιότητες με τις σχεσιακές, οι οποίες ουσιαστικά αποθηκεύουν τα δεδομένα τους σε σειρές (rows).

Μπορούν να χρησιμοποιούν SQL-τύπου γλώσσες για ερωτήματα ή μπορούν να έχουν κεντρικό ρόλο (backbone) σε ένα σύστημα για ETL processing (extract, transform, load επεξεργασία) και data visualization εργαλεία, όπως και οι σχεσιακές (row-oriented). Εντούτοις, μια "column-oriented" βάση μπορεί να προσπελάσσει με περισσότερη ακρίβεια και χωρίς να σκανάρει και να απορρίπτει κομμάτια σειρών (rows). Η απόδοση (performance) των ερωτημάτων αυξάνεται συγκρίνοντας με τις σχεσιακές και ειδικά σε μεγάλο όγκο δεδομένων (big data sets).

Σύγκριση Σχεσιακών και NoSQL Βάσεων

Η παραπάνω ανάλυση των δύο διαφορετικών φιλοσοφιών των σχεσιακών και NoSQL βάσεων μάς οδηγεί αυτόματα στη σύγκρισή τους. Όπως θα δούμε, το κάθε μοντέλο είναι κατάλληλο και πιο αποδοτικό σε διαφορετικές περιπτώσεις προβλημάτων, γι' αυτό δεν υπάρχει σωστό και λάθος μοντέλο, αλλά μοντέλο αποτελεσματικότερο ανάλογα με το πρόβλημα που τίθεται. Παρακάτω αναλύουμε περισσότερο τις διαφορές τους.

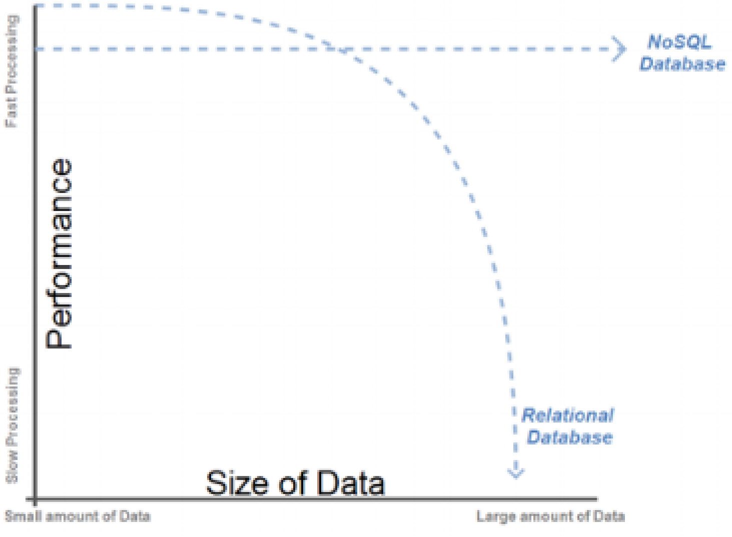
Οι δύο βασικές τους διαφορές εντοπίζονται στην οργάνωση, πώς οργανώνουν τη δομή των δεδομένων τους (schema), και η κλιμάκωση (scalability), οι σχεσιακές προσφέρονται καλύτερα για κάθετη κλιμάκωση, ενώ οι NoSQL για οριζόντια. Ας ξεκινήσουμε με την οργάνωση της δομής των δεδομένων (schema). Όπως αναφέραμε προηγουμένως, οι σχεσιακές βάσεις απαιτούν να έχει οριστεί το σχήμα δεδομένων (schema) προτού εισαχθούν στη βάση. Αυτό οφείλεται στην αυστηρή δομή των σχεσιακών βάσεων. Η γενική διάταξη της σχεσιακής βάσης, η οργάνωση των δεδομένων και οι σχέσεις μεταξύ αυτών πρέπει να γίνονται γνωστά σε πρώτο χρόνο. Αντιθέτως, στις NoSQL βάσεις κάτι τέτοιο είναι περιττό. Ακριβώς επειδή οι βάσεις αυτές χρησιμοποιούνται κατά κόρον πλέον για μεγάλα δεδομένα (big data), η συχνότητα δημιουργίας των οποίων είναι καθημερινή, το σχήμα των δεδομένων και η διάταξη της βάσης δεν γνωστοποιούνται εξ αρχής. Αυτό τις καθιστά κατάλληλες για μεγάλα δεδομένα, ενώ τις σχεσιακές ακατάλληλες. Υπενθυμίζουμε ότι με τον όρο "μεγάλα δεδομένα" εννοούμε δεδομένα με δυναμική δομή (η λήψη γίνεται τις περισσότερες αν όχι όλες τις φορές με διαφορετική δομή από ότι την προηγούμενη φορά) και μέγεθος τουλάχιστον petabytes. Συνεχίζουμε σχετικά με την κλιμάκωση που προσφέρουν οι σχεσιακές και NoSQL βάσεις και επανερχόμαστε στην ανάλυση μεγάλων δεδομένων (big data). Η χρήση NoSQL βάσεων φαίνεται καταλληλότερη, καθώς προσφέρουν οριζόντια κλιμάκωση (horizontal scaling), αποδοτικότερη για επεξεργασία μεγάλου όγκου δεδομένων σε λιγότερο κόστος και χρόνο. Η κάθετη κλιμάκωση (vertical scaling) των σχεσιακών είναι ακριβότερη και προορίζεται για συγκεκριμένη δόμηση δεδομένων. Δίνοντας τους δύο ορισμούς της κάθετης και οριζόντιας κλιμάκωσης μπορούμε να κατανοήσουμε καλύτερα τις διαφορές αυτών:

- Κάθετη κλιμάκωση (Vertical scaling): περιλαμβάνει τη βελτίωση των δυνατοτήτων ενός μόνο εξυπηρετητή (server), για παράδειγμα, χρησιμοποιώντας πιο δυνατή CPU, προσθέτοντας περισσότερη RAM, ή αυξάνοντας την χωρητικότητα

αποθήκευσης. Βέβαια, με βάση περιορισμούς της σύγχρονης τεχνολογίας των υπολογιστών, ένας μοναδικός server δεν καθίσταται συνήθως ικανός να αντιμετωπίσει μεγάλο φόρτο εργασίας. Ακόμη, οι πάροχοι νέφους (cloud providers) έχουν αυστηρά thresholds για χρήση διαθέσιμου hardware. Συνεπώς, από όλα τα παραπάνω βλέπουμε ότι πρακτικά υπάρχει μέγιστο όριο (φράγμα) για κάθετη κλιμάκωση.

- Οριζόντια κλιμάκωση (Horizontal scaling): περιλαμβάνει τη διαίρεση των δεδομένων σε πολλαπλούς servers, προσθέτοντας επιπλέον servers για αύξηση της χωρητικότητας και των δυνατοτήτων του συστήματος (συμπλέγματος – cluster). Εδώ, κάθε εξυπηρετητής μπορεί να μην έχει σημαντικά υψηλή χωρητικότητα, εντούτοις, παρουσιάζει συνήθως καλύτερη απόδοση από έναν ισχυρότερο σε πόρους server κάθετης κλιμάκωσης καθώς διαχειρίζεται ένα κλάσμα του φόρτου εργασίας. Ακόμη, η επέκταση των δυνατοτήτων του συστήματος απαιτεί απλά πρόσθεση περισσότερων servers στο cluster, που είναι φθηνότερο από καινούργιο και πιο εξελιγμένο hardware που απαιτεί ο μοναδικός εξυπηρετητής της κάθετης κλιμάκωσης. Το trade off εδώ είναι η πολυπλοκότητα της υποδομής και η συντήρηση.

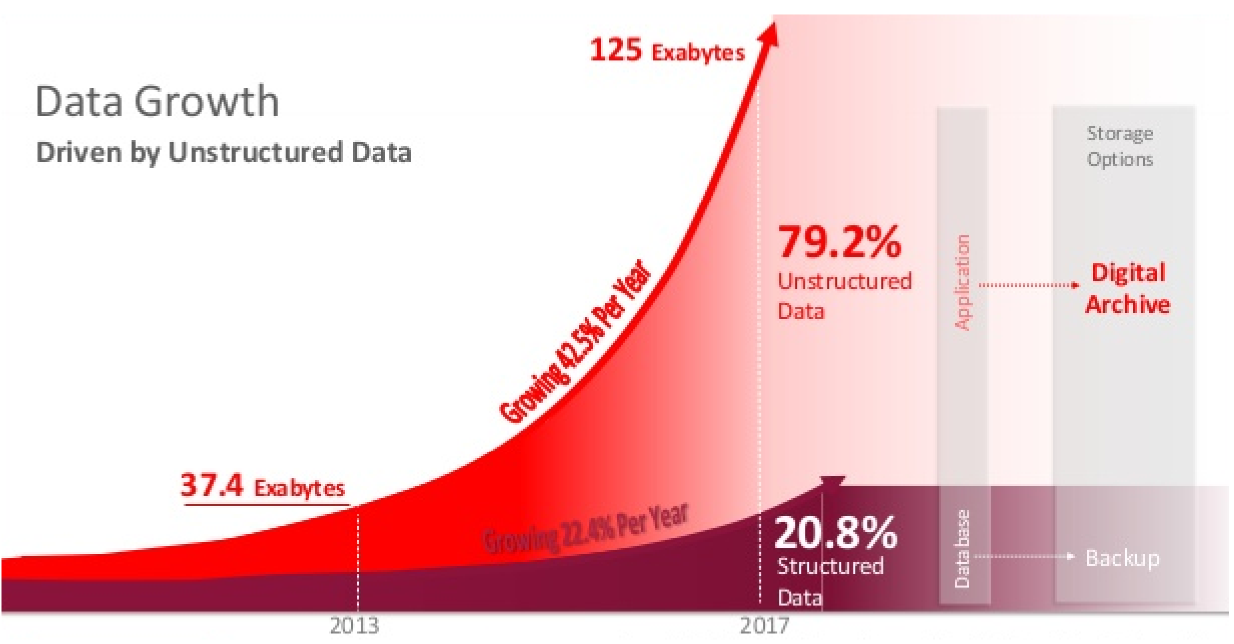
Scalability of **NoSQL** Database vs Traditional Relational Database



Εικόνα 7: Οι NoSQL είναι συνεπείς στην απόδοση όσο αυξάνεται το μέγεθος των δεδομένων ενώ τα RDBMS χάνουν σε ταχύτητα επεξεργασίας.

ACID

Στη συνέχεια, παραθέτουμε το κριτήριο ACID που ισχύει καλύτερα για τις σχεσιακές βάσεις και το θεώρημα CAP που είναι βασικός πυλώνας των NoSQL βάσεων. Οι σχεσιακές βάσεις δεδομένων οφείλουν την αυστηρή δόμηση των δεδομένων τους στην τήρηση του κριτηρίου ACID (Atomicity, Consistency, Isolation, Durability). Πρόκειται για ένα σύνολο ιδιοτήτων συναλλαγών βάσεων δεδομένων που εγγυώνται εγκυρότητα δεδομένων και συναλλαγών ακόμη και σε καταστάσεις σφαλμάτος του συστήματος ή διακοπής ρεύματος. Μια συναλλαγή βάσης δεδομένων (database transaction) είναι μια λογική διεργασία πάνω σε δεδομένα, όπως για παράδειγμα, η μεταφορά ποσού από έναν λογαριασμό σε άλλον θεωρείται συναλλαγή βάσης δεδομένων, παρά τα πιθανά ενδιάμεσα στάδια. Οι ιδιότητες αυτές είναι Ατομικότητα συναλλαγής (είτε πραγματοποιηθεί είτε όχι η συναλλαγή λόγω σφαλμάτων ή διακοπές ρεύματος, το σύστημα δεν θα επηρεαστεί), Συνοχή συναλλαγής (η κάθε συναλλαγή οδηγεί το σύστημα από τη μια έγκυρη κατάσταση στην άλλη), Απομόνωση συναλλαγής (μια αποτυχής συναλλαγή δεν επηρεάζει μια επόμενη) και Ανθεκτικότητα συναλλαγής (όταν πραγματοποιηθεί μια συναλλαγή καταγράφεται ως έγκυρη ακόμα και αν αμέσως μετά εκδηλωθεί κάποιο σφάλμα στο σύστημα – χρήση μη-πτητικής μνήμης). Όλη αυτή η ανάλυση έγινε από τη μία για να εξηγήσουμε τη λογική των σχεσιακών βάσεων και από την άλλη για να δείξουμε ότι όλοι αυτοί οι κανόνες είναι μεν σωστοί για τις περιπτώσεις εφαρμογής τους αλλά δεν είναι κατάλληλοι για το μεγάλο όγκο άγνωστης δομής δεδομένων που αντιμετωπίζουμε στη σύγχρονη εποχή.



Εικόνα 8: Τα Big Data αυξάνονται εκθετικά με την πάροδο του χρόνου σε σχέση με τα δομημένα δεδομένα που επεξεργάζονται τα συστήματα RDBMS.

Οι απαιτήσεις για επεξεργασία big data (unstructured data) ανεβάζουν τον πήχη και έχουν αφήσει πίσω κατά πολύ τις απαιτήσεις των δομημένων δεδομένων (structured data). Όπως φαίνεται στην παραπάνω εικόνα του Oracle Data Mining blog[18], ο ρυθμός αύξησης των αδόμητων δεδομένων είναι σχεδόν διπλάσιος από αυτόν των δομημένων δεδομένων. Σε αυτό το σημείο έρχονται να κάνουν την εμφάνισή τους οι NoSQL βάσεις που αποδεικνύονται κατάλληλες για μεγάλου όγκου (volume) συχνά (velocity) και ποικίλα (variety) δεδομένα (σύγχρονα big data), καθώς προσφέρουν ευελιξία, ταχύτητα και αποδοτικούς κατανεμημένους τρόπους αποθήκευσης (horizontal scaling). Οι NoSQL βάσεις σε συνδιασμό με το Cloud Computing αποτελούν το καλύτερο σύγχρονο εργαλείο για χειρισμό Big Data.

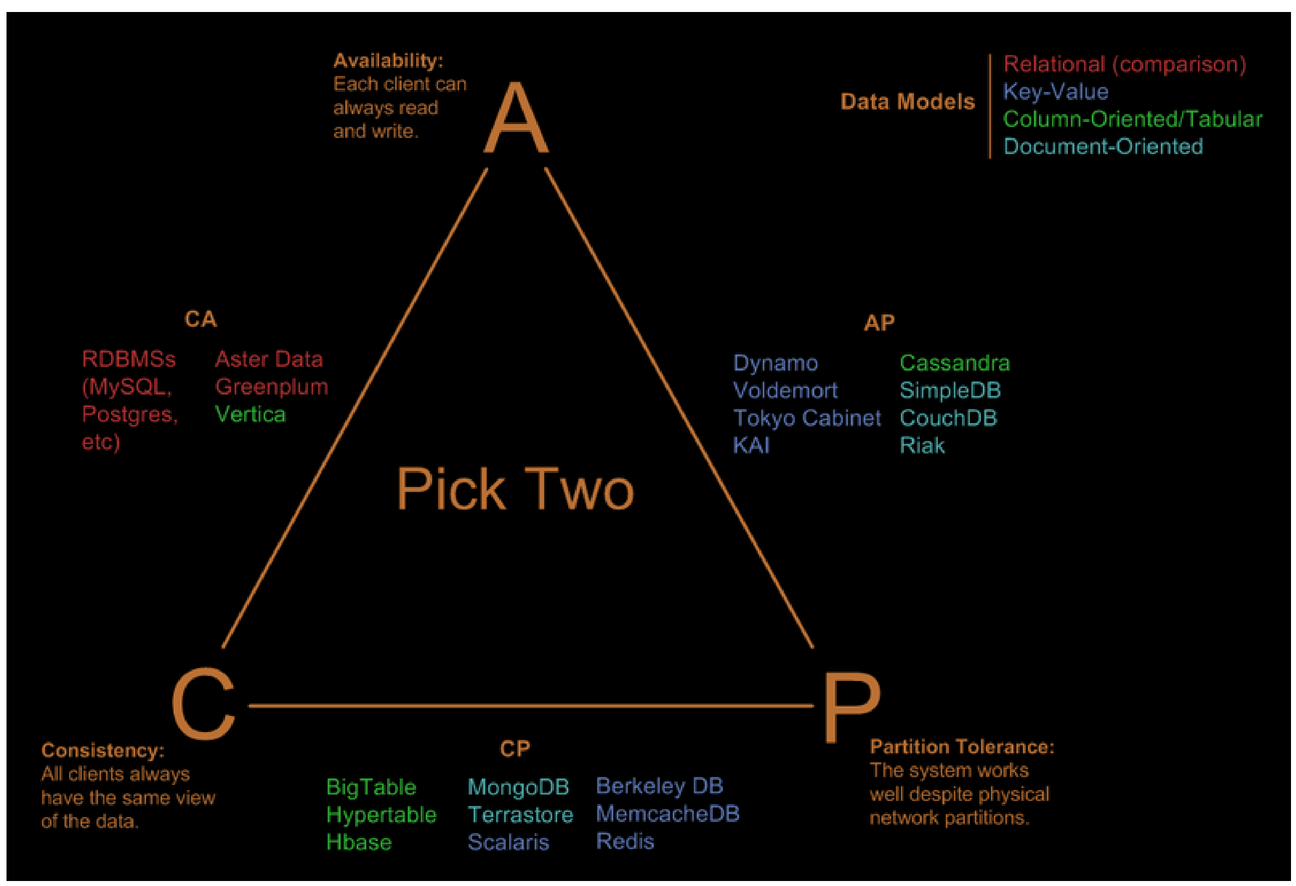
Οι ιδιότητες ACID δεν ικανοποιούνται πάντοτε από τις NoSQL βάσεις δεδομένων και αυτός δεν ήταν κάποιος στόχος προς επίτευξη εξ αρχής. Καταλήγουμε πλέον ότι οι NoSQL βάσεις έρχονται να προσφέρουν κάτι διαφορετικό και αυτό θα το κατανοήσουμε καλύτερα μέσα από το θεώρημα CAP.

Θεώρημα CAP

Το θεώρημα CAP ή Θεώρημα του Brewer, που διατυπώθηκε από τον Eric Brewer στο University of California, Berkeley, πρεσβεύει ότι σε κάθε κατανεμημένο σύστημα δεδομένων μπορούν να συνυπάρχουν το πολύ δύο από τα τρία επόμενα επιθυμητά χαρακτηριστικά:

- Συνέπεια (**C**onsistency) : όλοι οι πελάτες πάντα βλέπουν τα ίδια δεδομένα.
- Διαθεσιμότητα (**A**vailability): κάθε αίτηση σε έναν κόμβο στο σύστημα επιστρέφει μια απάντηση.
- Ανοχή διαχωρισμού (**P**artition tolerance): το κατανεμημένο σύστημα λειτουργεί παρά πιθανές καθυστερήσεις ή απορρίψεις μηνυμάτων ανάμεσα στους κόμβους του

Στην παρακάτω εικόνα φαίνεται πώς διαχωρίζει το CAP Theorem τα RDBMS συστήματα (κατηγορία CA) με πασίγνωστες NoSQL βάσεις, όπως οι MongoDB, Hbase, BigTable (CP), Cassandra, CouchDB, Riak (AP), και τις ίδιες τις NoSQL μεταξύ τους. Επομένως, παρατηρούμε ότι δεν υπάρχει καλύτερο σύστημα διαχείρισης βάσεων δεδομένων, όπως και δεν υπάρχει ένα είδος προβλήματος big data. Για κάθε διαφορετικό πρόβλημα διαχείρισης μεγάλων δεδομένων (big data) μπορούμε να βρίσκουμε και να χρησιμοποιούμε το συστημά βάσεων δεδομένων που μας εξυπηρετεί καλύτερα.



Εικόνα 9: Θεώρημα CAP: πώς δημοφιλή συστήματα διαχείρισης βάσεων δεδομένων σχετίζονται με τα χαρακτηριστικά του θεωρήματος (κατηγορίες CA, CP & AP).

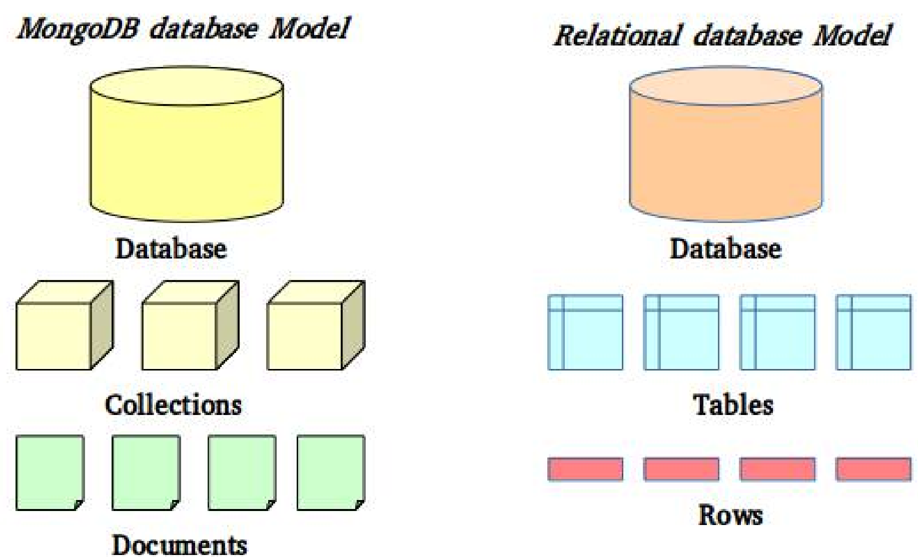
Η βάση δεδομένων MongoDB

Εικόνα 10: MongoDB

Εισαγωγή

Η MongoDB (από το hu**mongo**us=τεράστιος) αποτελεί μια ανοιχτού κώδικα εγγραφο-κεντρική (document-based) βάση δεδομένων που χρησιμοποιείται σε διαφορετικά λειτουργικά συστήματα

και ανήκει στην κατηγορία των NoSQL βάσεων, όπως είδαμε προηγουμένως. Αναπτύσσεται από την εταιρία MongoDB Inc. (με παλαιότερο όνομα 10gen) εδώ και χρόνια και χρησιμοποιείται από εταιρίες μεγαθήρια όπως Cisco, Google, Facebook, Ebay, Pearson, Adobe, Nokia, Verizon, Forbes κ.ά. Αποθηκεύει τα δεδομένα σε binary μορφή JSON εγγράφων (JavaScript Object Notation documents) που ονομάζονται BSON (Binary JSON). Συνήθως τα έγγραφα με παρόμοια μορφή οργανώνονται σε συλλογές (collections). Αν θέλαμε να κάνουμε μια αντιστοίχιση με τις σχεσιακές βάσεις θα λέγαμε ότι οι συλλογές είναι οι πίνακες (tables) και τα έγγραφα είναι οι εγγραφές (records) που στην περίπτωση των RDBMS αυτές οι εγγραφές βρίσκονται διασκορπισμένες σε διαφορετικούς πίνακες, ενώ εδώ τα έγγραφα είναι οργανωμένα σε μία συλλογή.



Εικόνα 11: MongoDB και RDBMS Models

Ας δούμε τώρα ένα έγγραφο BSON της MongoDB:

```
{
   name: "sue",      ←— field: value
   age: 26,          ←— field: value
   status: "A"       ←— field: value
}                                      } document
```

Εικόνα 12: BSON MongoDB document.

Παρατηρούμε ότι η MongoDB δομεί τα δεδομένα της σε ζεύγη τύπου "πεδίο:τιμή" (field:value) και ομαδοποιώντας τέτοια ζεύγη ορίζει ένα BSON document. Τα έγγραφα στη MongoDB αποθηκεύονται σε συλλογές (collections), όπως είδαμε, και προσφέρουν κι άλλες χρήσιμες ιδιότητες. Οι τιμές των πεδίων των BSON αντικειμένων μπορούν να πάρουν ως τιμές άλλα

έγγραφα, πίνακες ακόμα και πίνακες από έγγραφα και το δυναμικό τους σχήμα καθιστά δυνατό τον πολυμορφισμό (documents με διαφορετικές δομές στην ίδια collection). Παρακάτω συνεχίζουμε με βασικά χαρακτηριστικά που προσφέρει η βάση MongoDB.

Σημαντικά χαρακτηριστικά της βάσης MongoDB

Η MongoDB προσφέρει τα πλεονεκτήματα των NoSQL βάσεων μαζί με αυτά των document-based NoSQL βάσεων. Κάποιες βασικές ιδιότητες και δυνανότητες της αρχιτεκτονικής της είναι η οριζόντια κλιμάκωση (horizontal scaling), η υψηλή διαθεσιμότητα (high availability) και το υψηλό throughput, τα οποία προσφέρονται αντίστοιχα από τον θρυμματισμό (sharding), τα αντίγραφα που δημιουργεί (replica sets) αλλά και την κατανομή του φόρτου εργασία σε όλο το σύμπλεγμα (cluster load balancing). Παρακάτω αναφέρονται τα τρία διαφορετικά είδη κόμβων της MongoDB, αφού προηγηθεί η εξήγηση του sharding και των replica sets:

Sharding: Με τη μέθοδο αυτή, η MongoDB χωρίζει (θρυμματίζει) τα δεδομένα σε επίπεδο συλλογής (collection) χρησιμοποιώντας ένα κλειδί ονομαζόμενο "shard key". Τα κομμάτια στα οποία η MongoDB χωρίζει τα δεδομένα με βάση το shard key λέγονται "chunks". Μετά τον διαχωρισμό, τα διανέμει ανάμεσα στους "shard" κόμβους του cluster με τη βοήθεια του balancer. Ο balancer ελέγχει αν τηρείται εξισορρόπηση των chunks μιας συλλογής ανάμεσα στους shard κόμβους, και σε διαφορετική περίπτωση μετακινεί ένα chunk κάθε φορά ανάμεσα στους κατάλληλους shard κόμβους ώστε να πετύχει εξισορρόπηση. Ακόμη, είναι σημαντικό να αναφέρουμε τους τρόπους και τους περιορισμούς επιλογής shard key. Το shard key μπορεί να αποτελείται από ένα πεδίο (single index: "{name:1}") ή περισσότερα πεδία (compound index: "{name:1, price:1}"). Κάθε έγγραφο (document) της συλλογής που θρυμματίζεται (sharded collection) πρέπει να περιέχει το shard key. Οι τιμές του shard key διανέμονται στα κομμάτια (chunks) με μεθόδους όπως "Hashed sharding" (χρησιμοποιώντας hashed index σε ένα πεδίο του εγγράφου ως shard key), "Ranged sharding" (διαιρώντας με βάση το εύρος τιμών) ή "Tag Aware sharding" (ονομάζοντας το κάθε shard με ένα tag και διαχειρίζοντας τα shards πλέον μέσω των tags). Είναι φανερό ότι ανάλογα με το πρόβλημα που έχουμε κάθε φορά, επιλέγουμε και την μέθοδο που μας εξυπηρετεί καλύτερα.

Replica Sets (διατήρηση αντιγράφων): Με τη μέθοδο αυτή, η MongoDB προσφέρει υψηλή διαθεσιμότητα (high availability) και ανοχή σφαλμάτων (fault tolerance) καθώς διατηρεί αντίγραφα της ίδιας πληροφορίας σε διαφορετικούς κόμβους. Τα replica sets είναι ομάδες από mongod δαίμονες (διεργασίες) που αποτελούνται από έναν πρωτεύων κόμβο (primary node), ο οποίος λαμβάνει όλες τις write ενέργειες, και από έναν ή περισσότερους δευτερεύοντες κόμβους (secondary nodes), που ο καθένας τους αναπαράγει με την ίδια σειρά τις ενέργειες του πρωτεύοντος για να διατηρεί ταυτόσημα δεδομένα με αυτόν, αλλά χωρίς να επηρεάζει την

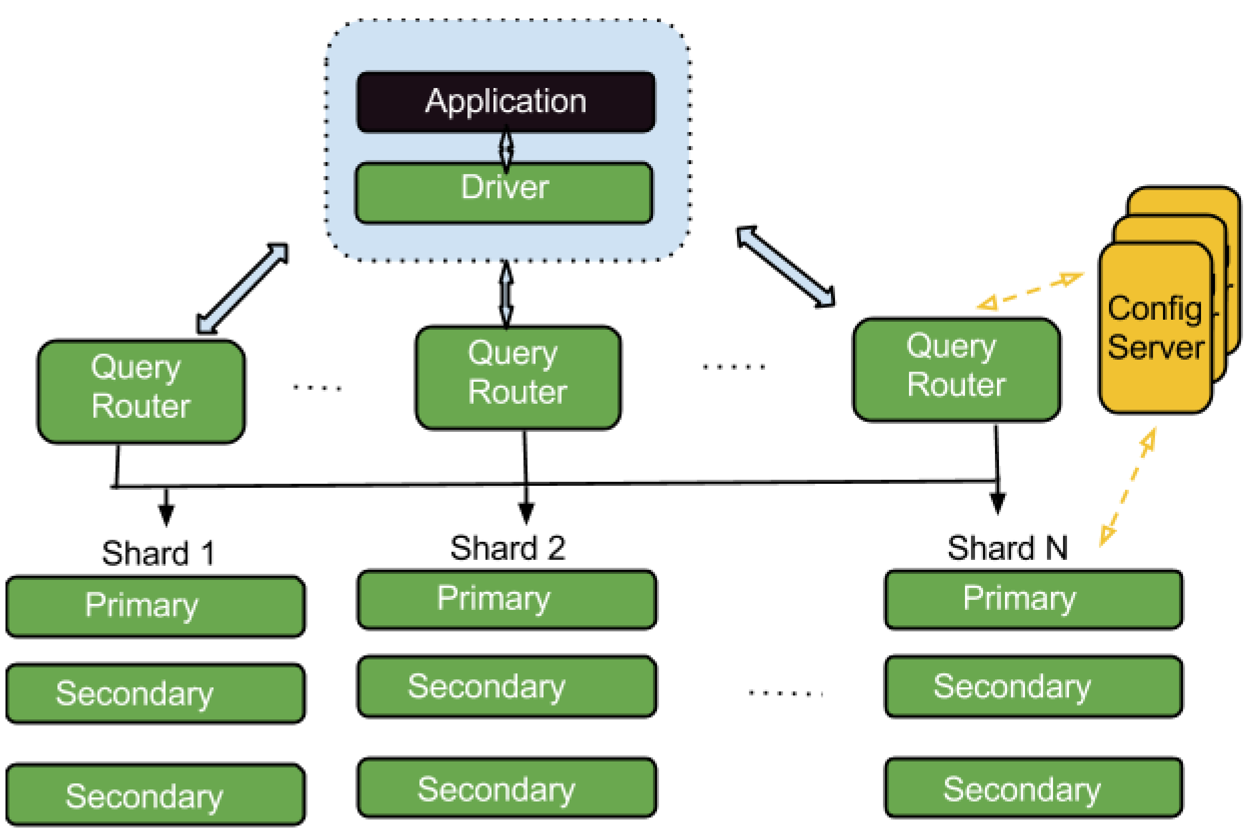
λειτουργία του. Κάθε εφαρμογή μπορεί να γράφει μόνο στον primary, αλλά έχει την επιλογή να διαβάζει είτε από αυτόν είτε από κάποιον secondary. Ακόμη, σε περίπτωση που ο primary γίνει για οποιοδήποτε λόγο μη διαθέσιμος, γίνονται εκλογές ανάμεσα στους secondary για να εκλέξουν νέο primary. Όταν ο προηγούμενος πρωτεύων γίνει και πάλι διαθέσιμος, συγχρονίζεται με τον νεό primary και συνεχίζει τη λειτουργία του πια ως secondary.

Παρακάτω συνεχίζουμε με την ανάλυση των κόμβων ως προς το είδος τους. Τα είδη είναι τρία:

Shard: ένας shard κόμβος αποθηκεύει ένα κλάσμα των δεδομένων του συμπλέγματος (cluster). Η πληροφορία όλων των shards αποτελεί το σύνολο των δεδομένων. Όπως φαίνεται στην παρακάτω εικόνα, ένα shard αντιστοιχεί σε ένα replica set που διατηρεί αντίγραφα των δεδομένων του shard.

Configuration Servers: αποθηκεύουν τα metadata ενός sharded cluster. Αυτά τα metadata περιλαμβάνουν την κατάσταση και την οργάνωση όλων των δεδομένων του cluster. Για παράδειγμα, τη λίστα των chunks που ανήκουν σε κάθε shard, τα εύρη τιμών που ορίζουν το κάθε chunk, κ.ά. Κάθε sharded cluster έχει τους δικούς του config servers και δεν τους μοιράζεται με άλλα clusters. Η βάση γράφει στους config servers όταν τα metadata αλλάζουν, για παράδειγμα, μετά από chunk migration ή chunk split, και διαβάζει από τους config servers όταν ξεκινάει κάποιο νέο mongos στιγμιότυπο ή γίνει αλλαγή στα metadata όπως αναφέραμε προηγουμένως.

Query Router: Σε ένα sharded cluster, είναι τα λεγόμενα "mongos" στιγμιότυπα (instances) που δρομολογούν τα queries και writes της εκάστοτε εφαρμογής στα shards. Για λόγους κυρίως ασφάλειας και απόδοσης, δεν επιτρέπεται στις εφαρμογές να έχουν άμεση επικοινωνία με τα shards. Ο κάθε query router ή mongos μαθαίνει ποια δεδομένα βρίσκονται σε ποια shards φέρνοντας στην cache τα metadata των configuration servers. Με αυτά τα metadata, δρομολογεί τα αιτήματα (queries, writes) των εφαρμογών στα κατάλληλα mongod στιγμιότυπα (shards). Με περισσότερους query routers έχουμε και καλύτερη ανταπόκριση του cluster στα αιτήματα της εφαρμογής.



Εικόνα 13: Αρχιτεκτονική MongoDB: οι query routers δρομολογούν τα αιτήματα στα κατάλληλα shards, κάθε shard είναι ένα replica set με primary και secondary κόμβους, κάθε κόμβος "τρέχει" έναν mongod δαίμονα.

Στη συνέχεια θα κάνουμε μια εισαγωγή στο Apache Spark, μια τελευταίας τεχνολογίας μηχανή επεξεργασίας δεδομένων σε μεγάλη κλίμακα που χρησιμοποιείται κατά κόρον στη σύγχρονη επιστήμη των υπολογιστών και κατανεμημένων συστημάτων.

Apache Spark: Lightning-Fast Cluster Computing

Εισαγωγή

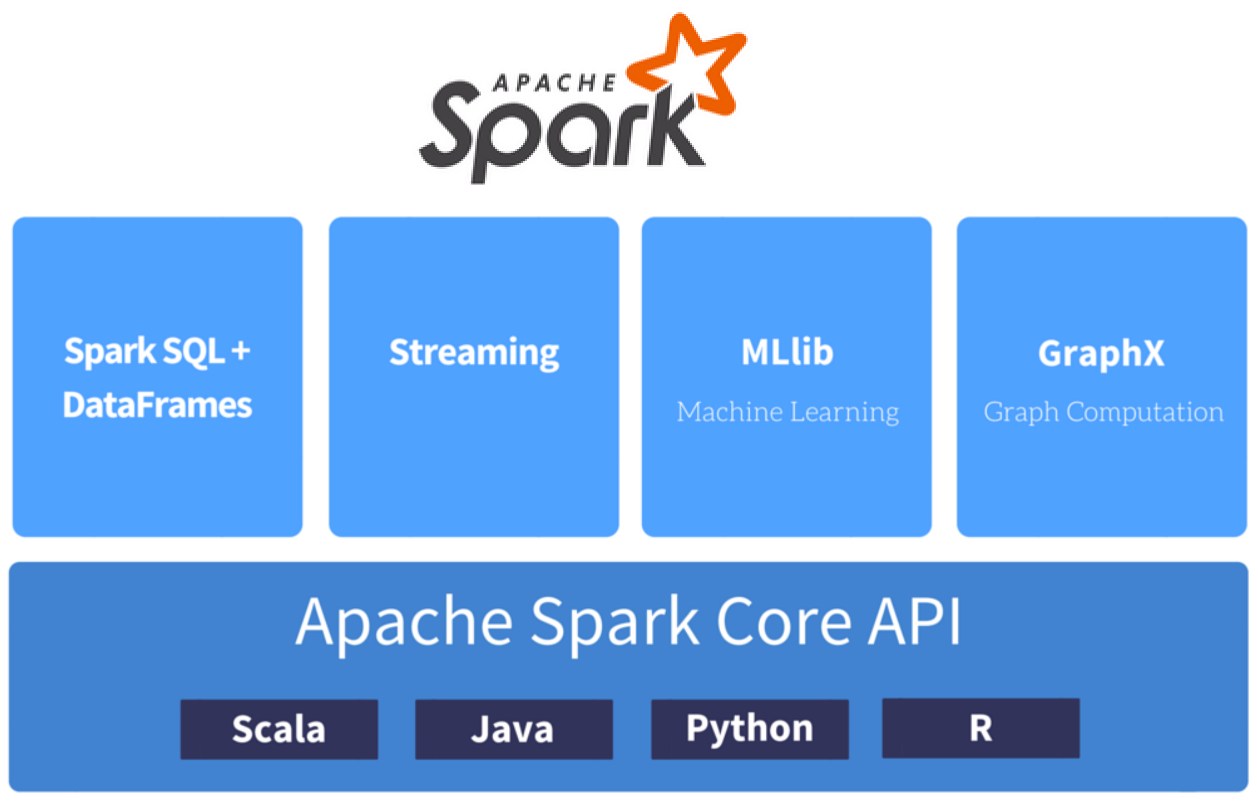
Το Apache Spark είναι μια ανοιχτού κώδικα και γενικού σκοπού μηχανή επεξεργασίας δεδομένων, ειδικά σχεδιασμένη για υψηλές ταχύτητες επεξεργασίας και ανάλυσης δεδομένων (big data). Το Spark είναι βασισμένο στο Hadoop MapReduce και επεκτείνει το μοντέλο του MapReduce εφόσον το χρησιμοποιεί αποδοτικότερα και σε διαφορετικού τύπου υπολογισμούς, όπως διαδραστικά ερωτήματα ή stream processing. Η βασικότερη ιδιότητα του Spark, που το καθιστά έως και 100 φορές πιο γρήγορο από το Hadoop MapReduce, είναι η παράλληλη in-memory επεξεργασία, δηλαδή επεξεργασία δεδομένων στη μνήμη, και όχι στο δίσκο, πράγμα που το απαλλάσει εξ ολοκλήρου από τις καθυστερήσεις των fetch του δίσκου. Ακόμη, είναι ειδικά σχεδιασμένο για την ταυτόχρονη υποστήριξη και διαχείριση διαφορετικών ειδών φόρτου εργασίας, όπως batch εφαρμογές, επαναλαμβανόμενους (iterative) αλγορίθμους, διαδραστικά ερωτήματα, streaming.

Ιστορική αναδρομή

Η πορεία του ως νέα τεχνολογία ξεκίνησε το 2009 σαν μικρό project του Hadoop στο AMPLab του UC Berkeley από τον Matei Zaharia. Έπειτα, έγινε επίσημα ανοιχτού κώδικα με BSD άδεια το 2010 και δωρήθηκε στην Apache Software Foundation το 2013. Από τον Φεβρουάριο του 2014 και μετά έγινε κορυφαίο project της εταιρίας και είναι από τα πιο δημοφιλή εργαλεία για big data analytics (ανάλυση μεγάλων δεδομένων). Εταιρίες μεγαθήρια όπως Amazon, eBay, Yahoo!, Netflix, κ.ά.[19] κάνουν εκτενή χρήση των δυνατοτήτων του Spark σε τεράστια cluster χιλιάδων κόμβων – το μεγαλύτερο γνωστό cluster περιέχει περισσότερους από 8000 κόμβους. Παρακάτω αναφέρουμε ενδεικτικά κάποια παραδείγματα. Το eBay αναλύει προσφορές, βελτιώνει την εξυπηρέτηση πελατών και τη γενική απόδοση του συστήματος χρησιμοποιώντας clusters των 2.000 κόμβων, 20.000 cores και 100 TB RAM μέσω YARN (διαχειριστής συμπλέγματος – θα εξηγηθεί παρακάτω). Ακόμη, με τη βοήθεια του Spark βελτιώθηκε η "News Personalization" (αρχική) σελίδα του Yahoo!. Ο παλιός αλγόριθμος μηχανικής μάθησης (machine learning) χρησιμοποιούσε 15.000 γραμμές κώδικα C++, ενώ τώρα με την Scala, που προσφέρει το Spark, ο αλγόριθμος μειώθηκε στις 120 γραμμές κώδικα και η εκπαίδευσή του διήρκεσε, σε συγκεκριμένο πείραμα, μόνο 30 λεπτά για 100 εκατομμύρια διαφορετικά σετ δεδομένων (datasets). Το Netflix χρησιμοποιεί Spark Streaming για να προσφέρει σε πραγματικό χρόνο online εξατομικευμένες προτάσεις στους πελάτες του. Επεξεργάζεται 450 δισεκατομμύρια "events" τη μέρα – ένα event συλλαμβάνει όλες τις δραστηριότητες των μελών και άρα παίζει βασικό ρόλο στην εξατομίκευση. Έπειτα από την αναφορά στην ιστορική του πορεία και σε σύγχρονα παραδείγματα χρήσης, συνεχίζουμε εξηγώντας καλύτερα την λειτουργία και την αρχιτεκτονική του Spark.

Ανάλυση Αρχιτεκτονικής Δομής

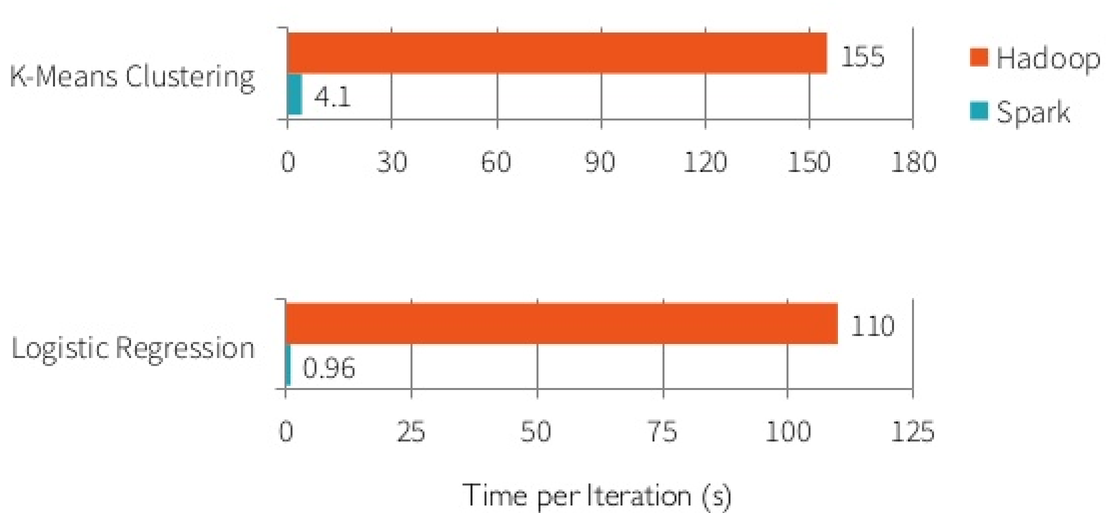
Το Spark είναι ένα ισχυρό, γρήγορο και ανεκτικό σε σφάλματα cluster υπολογιστικό σύστημα. Αποτελείται από τον πυρήνα, Spark Core, και ένα σετ βιβλιοθηκών. Στον πυρήνα γίνεται κατανεμημένη επεξεργασία δεδομένων (big data), ενώ προσφέρονται υψηλού επιπέδου APIs σε Scala, Java, Python και R για την ανάπτυξη κατανεμημένων ETL εφαρμογών (Extract, Transform, Load). Ακόμη, διαθέτει ένα σύνολο εργαλείων πιο υψηλού επιπέδου: το Spark SQL, που προσφέρει μια νέα αφαιρετική δομή δεδομένων, το RDD, κατάλληλο για SQL-τύπου ερωτήματα σε δομημένα και ημι-δομημένα δεδομένα, το Spark Streaming, το οποίο εκμεταλλεύεται την ταχύτατη δρομολόγηση του Spark Core για streaming ανάλυση, την MLlib (Machine Learning Library), μια κατανεμημένη machine learning πλατφόρμα πάνω από το in-memory κατανεμημένο Spark, και το GraphX, μια ειδική για επεξεργασία γράφων κατανεμημένη πλατφόρμα πάνω από το Spark.



Εικόνα 14: Apache Spark logo, components and APIs

Όπως αναφέραμε, ένα βασικό χαρακτηριστικό του Spark είναι η ταχύτητα, ενώ τη λίστα αυτή έρχονται να συμπληρώσουν η ευκολία χρήσης του και η ιδέα "Spark Ecosystem", δηλαδή ότι όλοι οι μηχανισμοί του προσφέρονται σε ένα ενοποιημένο πλαίσιο. Πιο συγκεκριμένα, η ταχύτητα επεξεργασίας και ανάλυσης μεγάλων δεδομένων (big data), ώντας το νούμερο ένα χαρακτηριστικό που έκανε δημοφιλές το Spark, σχετίζεται με τον χρόνο αναλυτικής εξέτασης των μεγάλων δεδομένων και τον χρόνο αναμονής μέχρι το τέλος της επεξεργασίας. Παρακάτω βλέπουμε την

απόδοσή του σε σχέση με το MapReduce του Hadoop σε δύο γνωστούς αλγορίθμους, τους k-means clustering και λογιστική παλινδρόμηση:



Εικόνα 15: Το Spark είναι σχεδόν 40 φορές πιο γρήγορο από το Hadoop MapReduce στον αλγόριθμο K-means clustering και πάνω από 100 φορές πιο γρήγορο στον αλγόριθμο λογιστικής παλινδρόμησης.

Αποτελώντας ένα από τα πιο γρήγορα υπολογιστικά συστήματα cluster για ανάλυση και επεξεργασία μεγάλων δεδομένων (big data analytics), το Apache Spark επεκτείνει το μοντέλο MapReduce του Hadoop καταφέρνοντας να εμφανιστεί έως και 100 φορές πιο γρήγορο. Ο βασικός λόγος, όπως αναφέραμε προηγουμένως, είναι η δυνατότητα μεταφοράς των υπολογισμών στη μνήμη (in-memory parallel processing), εξού και η τεράστια διαφορά σε επαναλήψεις στην παραπάνω εικόνα. Πιο σχολαστική ανάλυση του τρόπου υπολογισμού στη μνήμη θα γίνει σε λίγο στην εξήγηση της αφαιρετικής δομής RDD.

Ακόμη, πριν συνεχίσουμε με την ανάλυση του Spark Core (πυρήνας), είναι αναγκαίο να αναφέρουμε το κύριο πλεονέκτημα του εύχρηστου ενοποιημένου πλαισίου "Spark Ecosystem". Οποιαδήποτε βελτίωση σε χρόνο συμβαίνει σε κάποιο API ή στον πυρήνα, αυτόματα μεταφράζεται ως καλύτερη απόδοση και στα πιο πάνω επίπεδα, όπως οι βιβλιοθήκες, η Spark SQL, το streaming. Με αυτόν τον τρόπο, οποιοσδήποτε χρησιμοποιεί την πλατφόρμα, μπορεί άμεσα να επωφεληθεί από κάποια καινούργια βελτίωση, χωρίς να κρίνεται απαραίτητη η επανεγκατάσταση του όλου συστήματος.

Spark Core

Ο πυρήνας αποτελεί τον σύνδεσμο όλων των λειτουργιών και ιδιοτήτων της πλατφόρμας. Προσφέρει κατανεμημένα διαμοιρασμό εργασιών, δρομολόγηση και λειτουργίες εισόδου και εξόδου, μέσω διεπαφών APIs με κέντρο την αφαιρετική δομή RDD (Resilient Distributed Datasets). Διαχειριστική οντότητα της ανάλυσης δεδομένων αποτελεί ο driver, το πρόγραμμα οδήγησης που θα εκκινήσει διαδικασίες παράλληλης επεξεργασίας ενός RDD, περνώντας μια συνάρτηση στο Spark, το οποίο με τη σειρά του θα δρομολογήσει την πραγματική παράλληλη επεξεργασία μέσα στο cluster.

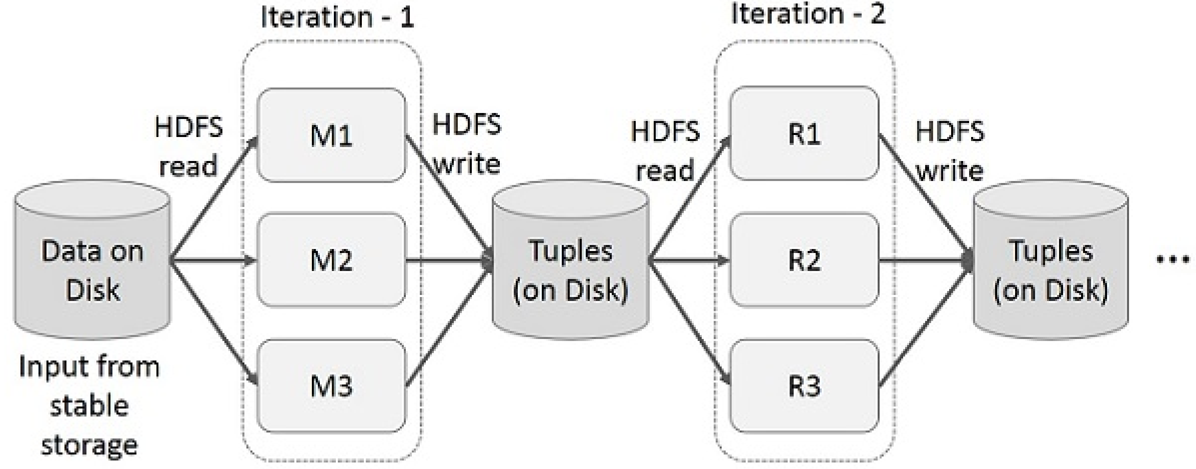
RDD – Resilient Distributed Datasets

RDD - Resilient Distributed Datasets[20]: Η θεμελιώδης δομή δεδομένων του Spark που αποτελεί ένα σύνολο αντικειμένων σε αφαιρετική δομή (datasets) ανεκτικό σε σφάλματα (resilient) κατάλληλο για παράλληλη επεξεργασία (distributed). Κάθε dataset μέσα σε ένα RDD χωρίζεται σε λογικά μέρη, το καθένα από τα οποία μπορεί να επεξεργάζεται σε διαφορετικούς κόμβους μέσα στο σύμπλεγμα (cluster). Ακόμη, κάθε RDD μπορεί να περιέχει αντικείμενα (objects) κάθε τύπου, για παράδειγμα, Scala, Java ή Python Object ή κλάσεις ορισμένες απο τον χρήστη.

Για τη δημιουργία ενός RDD, είτε πρέπει με τον driver να παραλληλοποιήσουμε μια υπάρχουσα συλλογή αντικειμένων (parallelize), είτε να εισάγουμε δεδομένα (big data) από εξωτερική πηγή (όπως HDFS) σε μορφή Hadoop Input Format. Το RDD αποτελεί τον τρόπο που το Spark κάνει ταχύτερη και αποδοτικότερη χρήση του μοντέλου MapReduce του Hadoop.

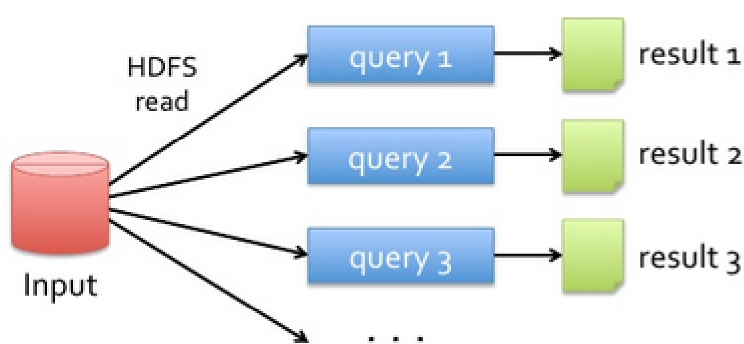
Σύγκριση με Hadoop MapReduce

Το Hadoop MapReduce χρησιμοποιείται κατά κόρον για παράλληλη επεξεργασία μεγάλων δεδομένων σε cluster. Προσφέρει διάφορους αφαιρετικούς τρόπους πρόσβασης στους υπολογιστικούς πόρους ενός cluster. Εντούτοις, εντοπίζεται αρκετά αργό πλέον έναντι του Spark, γιατί όλες οι διαδικασίες, είτε επαναληπτικές (iterative) είτε διαδραστικές (interactive), απαιτούν άμεσο διαμοιρασμό των νέων υπολογισμένων δεδομένων σε πραγματικό χρόνο ανάμεσα σε παράλληλες εργασίες (data sharing). Ο μόνος τρόπος στο Hadoop MapReduce είναι μια εργασία να γράψει τα νέα δεδομένα σε κάποιον εξωτερικό αποθηκευτικό χώρο και η άλλη παράλληλη εργασία να τα διαβάσει από εκεί – το I/O του δίσκου αργοπορεί από μόνο του, ενώ χρειάζεται πιθανώς σειριοποίηση επί πρόσθετα.



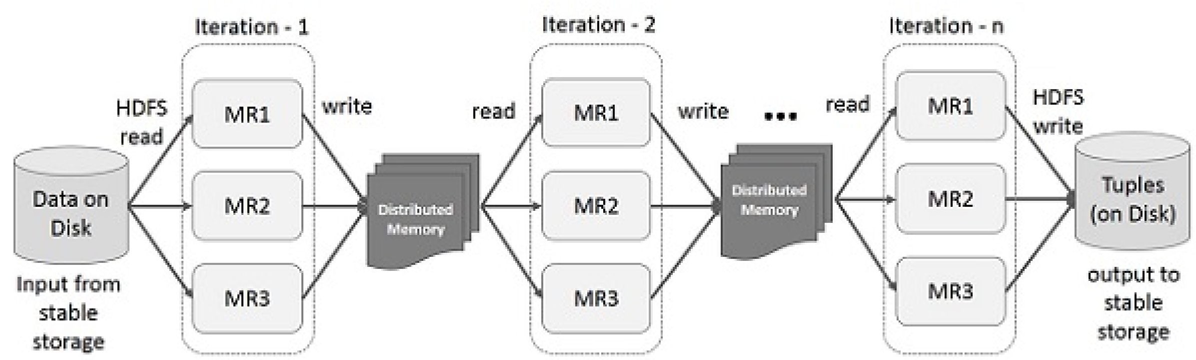
Εικόνα 16: Επαναληπτική διαδικασία στο Hadoop MapReduce.

Τα reads και τα writes στο δίσκο δεν είναι αποδοτικά σε χρόνο, ειδικά όταν η επόμενη επανάληψη (iteration) χρειάζεται τα αποτελέσματα της προηγούμενης σε πραγματικό χρόνο.



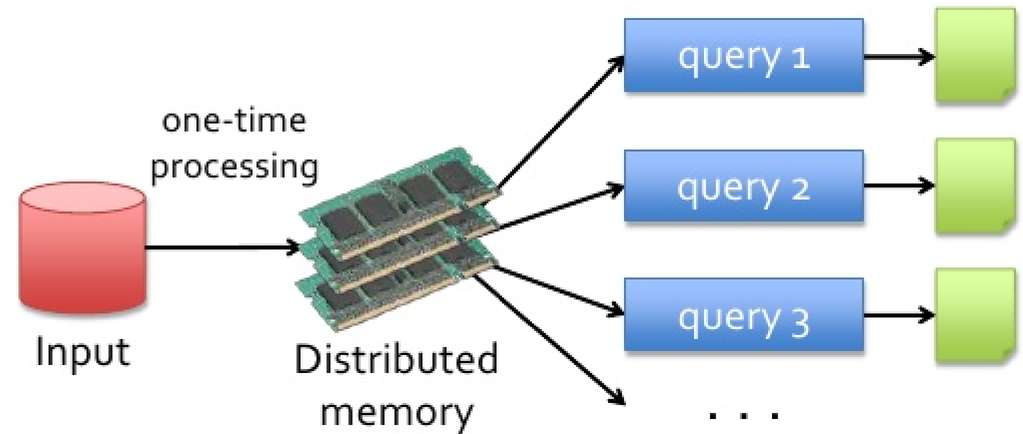
Εικόνα 17: Διαδραστική διαδικασία στο Hadoop MapReduce. Για κάθε query (ερώτημα) χρειάζεται και ένα read από τον δίσκο.

Το πρόβλημα της καθυστέρησης των σειριοποίησης, I/O δίσκου και replication λύνει το Spark με το RDD. Τώρα δεν χρειάζεται στα ενδιάμεσα δεδομένα που δημιουργούνται να αποθηκεύονται στον δίσκο για να διαβαστούν από την επόμενη επανάληψη. Το Spark προσφέρει in-memory υπολογισμό:



Εικόνα 18: Το Spark αποθηκεύει τα αποτελέσματα της κάθε επανάληψης σε κατανεμημένη μνήμη, έτσι η ανάκτησή τους γίνεται σε ταχύτητα RAM, και όχι δίσκου.

Εάν τα αποτελέσματα των ενδιάμεσων σταδίων δεν χωράνε εξ ολοκλήρου στην κατανεμημένη RAM, θα αποθηκευτούν στο δίσκο. Αλλά κάτι τέτοιο δεν συμβαίνει συχνά, αφού η RAM επεκτείνεται, όταν χρειάζεται, με τη χρήση νέων κόμβων και μετράται σε εκατοντάδες terabytes, όπως είδαμε προηγουμένως και στο παράδειγμα του eBay. Συνεχίζουμε με παράδειγμα του Spark για διαδραστική διαδικασία:



Εικόνα 19: Διαδραστική διαδικασία στο Spark.

Για κάθε query (ερώτημα) χρειάζεται read από τη μνήμη, πολύ πιο γρήγορο από το δίσκο, εφόσον γίνουν fetch τα δεδομένα από το δίσκο στη μνήμη.

Σε αυτή τη διαδικασία, το Spark "φέρνει" στη μνήμη το κατάλληλο dataset και θέτει τα επιθυμητά ερωτήματα. Τα αποτελέσματα των ερωτημάτων, ένα νέο RDD, μπορούν να γίνουν "persist" στη μνήμη, δηλαδή να μείνουν στη μνήμη για μετέπειτα γρηγορότερη πρόσβαση και επεξεργασία, κάτι που δεν συμβαίνει στην περίπτωση του Hadoop MapReduce.

Lazy Evaluation

Η υπεροχή των τεχνολογιών του Spark δεν σταματάει στη χρήση RAM έναντι δίσκου, την οποία αναλύσαμε. Ο τρόπος χειρισμού της αφαιρετικής δομής RDD είναι κλειδί. Οι ενέργειες που μπορούν να γίνουν πάνω σε ένα RDD είναι είτε μετασχηματισμοί (transformations), οι οποίοι δημιουργούν ένα νέο σύνολο αντικειμένων (RDD) από ένα ήδη υπάρχον, είτε δράσεις (actions), οι οποίες εκτελούν έναν υπολογισμό πάνω στο RDD και επιστρέφουν το αποτέλεσμα (μία τιμή) στον driver του Spark. Παρακάτω αναφέρουμε ως παραδείγματα τις συναρτήσεις map() και reduce(), ονόματα εννοιών που μας έχουν απασχολήσει πολύ μέχρι στιγμής σχετικά με την λειτουργία του Spark και τις διαφορές του με το Hadoop MapReduce. Η συνάρτηση map() είναι ένας μετασχηματισμός που επιστρέφει ένα νέο κατανεμημένο RDD περνώντας όλα τα δεδομένα (big

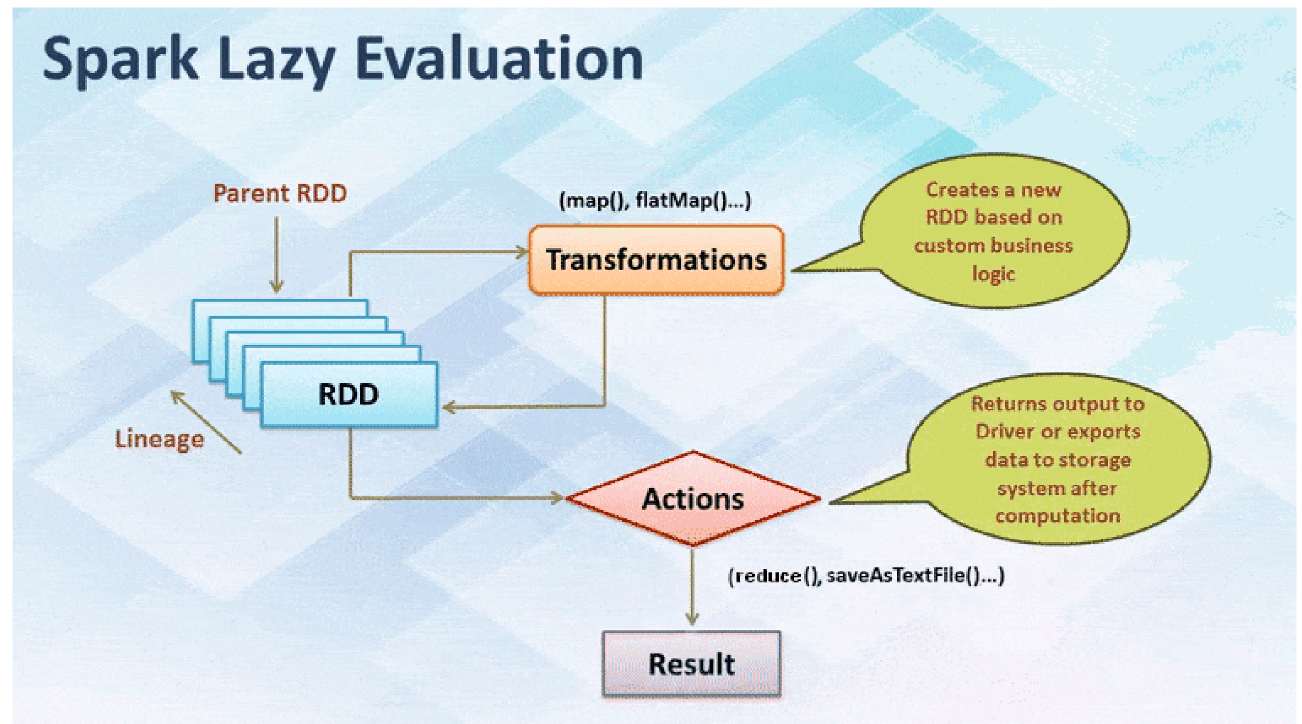
data) του RDD εισόδου μέσα από μια συνάρτηση. Ακόμη, η συνάρτηση reduce() είναι μια δράση που συγκεντρώνει όλα τα δεδομένα ενός RDD χρησιμοποιώντας μια αντιμεταθετική και προσεταιριστική συνάρτηση (για να είναι δυνατός ο υπολογισμός παράλληλα) τύπου (a,b)→ c (εδώ το RDD εισόδου πρέπει να είναι κατάλληλο σε μορφή ζευγών (a,b)) και επιστρέφει το αποτέλεσμα στον driver. Το πλεονέκτημα χειρισμού των RDD εντοπίζεται στο γεγονός ότι όλοι οι μετασχηματισμοί (transformations) είναι "lazy", δηλαδή δεν εκτελούνται αμέσως πάνω στα δεδομένα (big data), αλλά αποθηκεύονται πάνω σε έναν γράφο υπολογισμού τύπου DAG (Directed Acyclic Graph of computation) όπου οργανώνονται και ομαδοποιούνται κατάλληλα. Η πραγματική εκτέλεση των μετασχηματισμών ενεργοποιείται με την εμφάνιση της πρώτης στη σειρά δράσης (action) που απαιτεί την επιστροφή ενός αποτελέσματος στον driver, όπως είδαμε. Η οκνηρή αποτίμηση (lazy evaluation) καθιστά το Spark πιο αποδοτικό καθώς:

- Βελτιώνεται η διαχειρισιμότητα: μέσω της lazy evaluation οι χρήστες μπορούν να οργανώσουν το πρόγραμμά τους σε μικρότερες διαδικασίες (βολικό για κάθε προγραμματιστή) καθώς ομαδοποιώντας τις ενέργειες στα μεγάλα δεδομένα (transformations ή actions) μειώνονται οι φορές αλληλεπίδρασης με το σύνολο των δεδομένων.

- Μείωση υπολογισμών και αύξηση ταχύτητας: μέσω της lazy evaluation μειώνεται το overhead των υπολογισμών. Υπολογίζονται μόνο οι απαραίτητες τιμές, την κατάλληλη στιγμή, χωρίς να χρειάζεται να γνωστοποιούνται στον driver ενδιάμεσα αποτελέσματα εκτελώντας ανούσιες μεταφορές δεδομένων – γρηγορότερη και αποδοτικότερη επεξεργασία.

- Μείωση πολυπλοκότητας: Από τη στιγμή που το Spark δεν εκτελεί κάθε υπολογισμό λόγω lazy evaluation, κερδίζει σε χρόνο.

- Βελτιστοποίηση ερωτημάτων: το Spark μειώνει τον αριθμό των ερωτημάτων (queries) πάνω στο RDD.

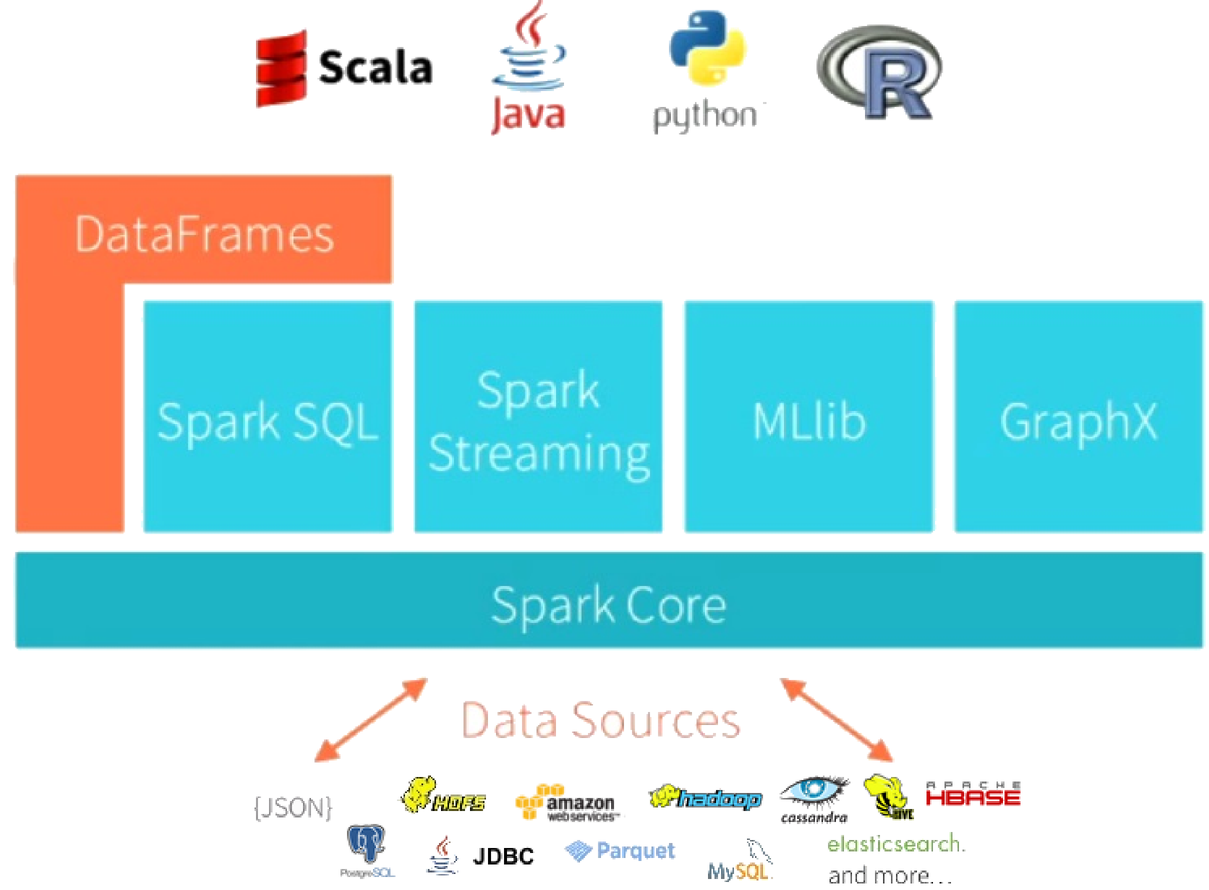


Εικόνα 20: Spark Lazy Evaluation

Μας μένουν άλλα δύο σημαντικά κομμάτια της λειτουργίας του Spark: οι βιβλιοθήκες που προσφέρει και οι διαχειριστές συμπλέγματος (cluster managers) που υποστηρίζει. Συνεχίζουμε αναφέροντας τις βιβλιοθήκες που προσφέρουν εργαλεία υπολογισμού σε διαφορετικούς τομείς της επιστήμης των υπολογιστών.

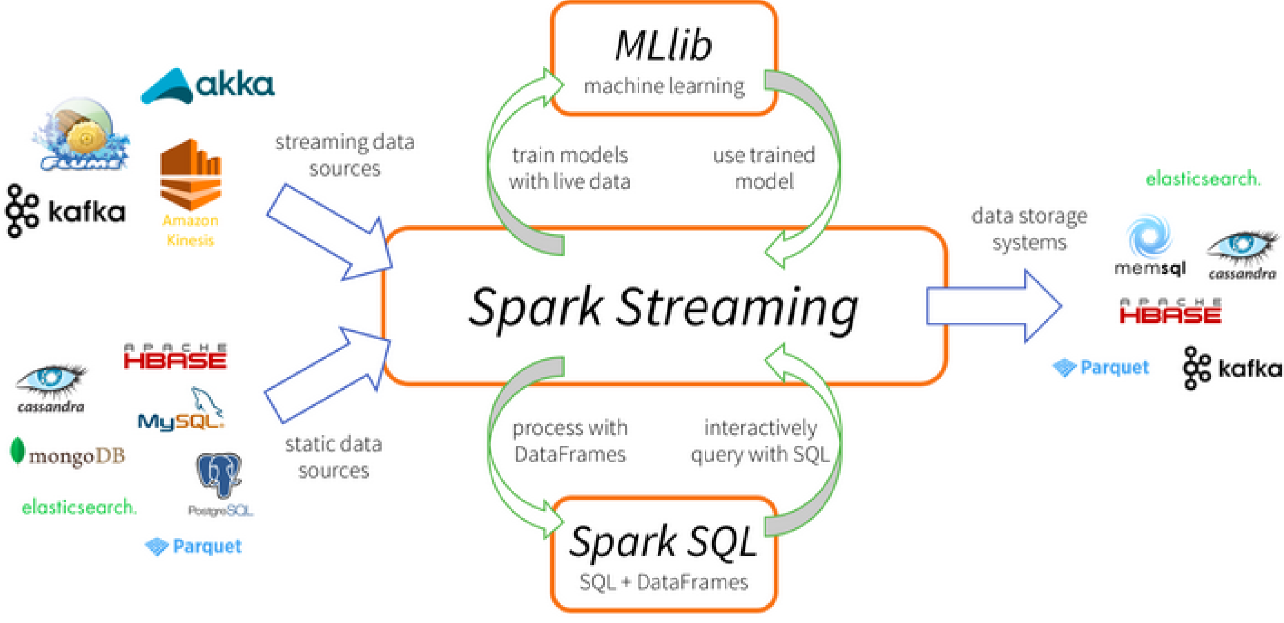
MLlib ή Machine Learning Library. Η MLlib είναι η κλιμακώσιμη, μηχανικής μάθησης βιβλιοθήκη του Spark. Σκοπός της είναι να κάνει τη μηχανική μάθηση εύκολη και κλιμακώσιμη (scalable), προσφέροντας υψηλού επιπέδου αλγορίθμους, όπως ταξινόμησης, παλινδρόμησης, ομαδοποίησης, συνδυαστικού φιλτραρίσματος. Ακόμη, είναι συμβατή με εφαρμογές σε Scala, Java ή Python.

Spark SQL. Με το Spark SQL προσφέρεται η δυνατότητα για επεξεργασία δομημένων δεδομένων (big data). Είναι κατάλληλο για την διατύπωση SQL-τύπου ερωτημάτων στην αφαιρετική δομή δεδομένων RDD και υποστηρίζει Scala, Java, Python και R. Διαθέτει την αφαιρετική δομή των λεγόμενων dataframes, η οποία επιτρέπει την πρόσβαση, ερώτηση αλλά και ένωση (join) με τον ίδιο τρόπο διαφορετικών δομών δεδομένων, όπως Hive, Avro, Parquet, ORC, JSON και JDBC.



Εικόνα 21: Spark SQL αρχιτεκτονική

Spark Streaming. Όπως φανερώνει το όνομά του, το Spark Streaming καθιστά την πλατφόρμα ικανή να επεξεργάζεται ροές δεδομένων (big data streaming). Με το Spark Streaming μπορούν να κατασκευαστούν κλιμακώσιμες ανεκτικές σε σφάλματα εφαρμογές επεξεργασίας ροώς δεδομένων με απαιτήσεις πραγματικού χρόνου. Στην παρακάτω εικόνα βλέπουμε ότι το Spark Streaming μπορεί να δέχεται μεγάλα δεδομένα από στατικές πηγές, όπως πληροφορίες ατόμων σε βάσεις δεδομένων εταιριών, κ.ά., καθώς και δυναμικές πηγές μεγάλων δεδομένων, όπως δημοσιεύσεις σε κοινωνικά δίκτυα, καιρός, εφαρμογή του k-means για εύρεση καλύτερης διαδρομής σε χάρτη κ.ά. Διαθέτει κατάλληλο API για RDD, είναι συμβατό με Scala, Java και Python, ενώ προσφέρεται για διαδραστικά ερωτήματα ή εφαρμογές αλγορίθμων μηχανικής μάθησης σε δυναμικά δεδομένα (live streaming data – RDD μορφή).



Εικόνα 22: Αρχιτεκτονική του Spark Streaming σε συνδυασμό σε στατικές και δυναμικές πηγές δεδομένων.

GraphX. Η βιβλιοθήκη αυτή προσφέρει ευκολία στην επεξεργασία και τη διαχείριση γράφων (για παράδειγμα γράφος κοινωνικών δικτύων, γράφος οδικού χάρτη κ.ά.) αντιμετωπίζοντας τα δεδομένα (big data) είτε σαν συλλογή είτε σαν γράφο. Με τη λογική αυτή γίνονται αποδοτικά μετατροπές και ενώσεις γράφων, επεκτείνοντας το RDD API, όπως και οι παραπάνω βιβλιοθήκες, ενώ δίνεται η δυνατότητα να αναπτυχθούν αλγόριθμοι για γράφους. Το GraphX, εκμεταλλευόμενο την ταχύτητα του Spark, αποτελεί ήδη ένα από τα γρηγορότερα συστήματα επεξεργασίας γράφων, ενώ παρέχει γνώστους και χρήσιμους αλγορίθμους, όπως PageRank, Connected components, Label propagation, SVD++, Strongly connected components και Triangle count[21] – πολλοί από τους οποίους αναπτύχθηκαν από χρήστες.

Στη συνέχεια κλείνουμε την αναφορά μας στο Apache Spark με την περιγραφή των διαχειριστών συμπλέγματος (τί είναι και γιατί είναι χρήσιμοι), και βασικών ρυθμίσεων και αρχικοποιήσεων.

Διαχειριστές συμπλέγματος (Cluster Managers)

Μια Spark εφαρμογή είναι δυνατόν να εκτελεστεί είτε τοπικά, σε έναν μόνο υπολογιστή (local mode), είτε κατανεμημένα, σε ένα σύμπλεγμα υπολογιστών (2 modes: cluster ή client).

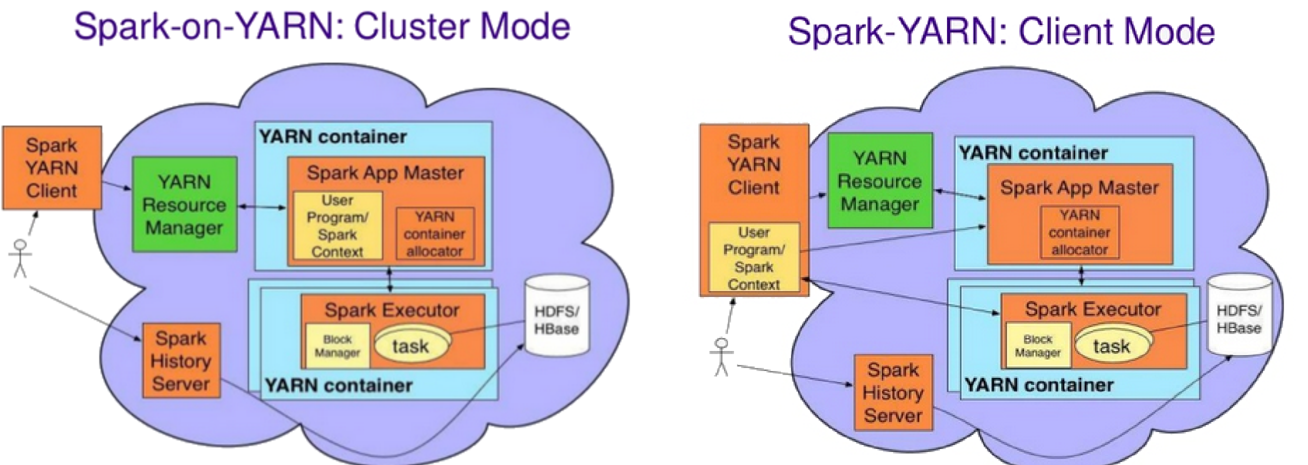
Για την περίπτωση εκτέλεσης της εφαρμογής σε local mode, δημιουργείται μόνο ένας JVM executor και μέσα σε αυτό υπάρχει ο driver (πρόγραμμα οδήγησης ή SparkContext). Μπορούμε να τρέξουμε την εφαρμογή με τρεις τρόπους: "local", για μόνο ένα thread, "local[2]", παράλληλα σε

δύο threads, ιδανικό εάν ο υπολογιστής μας έχει δύο φυσικούς πυρήνες, και "local[*]", για να τρέξει παράλληλα με τόσα threads, όσους λογικούς πυρήνες έχει ο υπολογιστής. Το local mode χρησιμοποιείται κυρίως για έλεγχο σφαλμάτων και άλλες δοκιμές, προτού επιχειρηθεί η εκτέλεση σε κατανεμημένο σύστημα υπολογιστών, και δεν απαιτεί καμία τροποποίηση ή ρύθμιση του Spark εκ των προτέρων.

Βέβαια, το Spark έχει σχεδιαστεί για μεγάλα συμπλέγματα χιλιάδων υπολογιστικών κόμβων. Για αυτές τις περιπτώσεις έχουμε δύο ειδών εκτελέσεις: cluster mode και client mode. Η διαφορά έγκειται στο περιβάλλον που τρέχει κάθε φορά ο driver (πρόγραμμα οδήγησης). Στο cluster mode, ο driver τρέχει μέσα στο σύμπλεγμα (cluster), ενώ στο client mode εξωτερικά από το cluster, σε κάποιον υπολογιστή. Το Spark χρησιμοποιεί αρχιτεκτονική τύπου master/slave για τους κόμβους του cluster (αντί για slave, χρησιμοποιείται η ορολογία worker), ενώ ο driver είναι η κύρια διεργασία μιας εφαρμογής που υποβάλλεται στο Spark και αυτός που επικοινωνεί με τους διαχειριστές συμπλέγματος. Οι cluster managers μπορεί να είναι ο Standalone Scheduler, εγκατεστημένος εξ αρχής μέσα στο Spark – απλή και εύκολη λύση για μικρά cluster, ο Hadoop YARN, και ο Apache Mesos, για production clusters, με τον Mesos να αποτελεί τον πλέον καλύτερο διαχειριστή πόρων. Υπάρχει ακόμη ένα project της Apache για scheduling που συνδυάζει τους Hadoop YARN και Apache Mesos, εν ονόματι Apache Myriad, αλλά ξεφεύγει από τα πλαίσια αυτής της ανάλυσης.

Αρχικά, ο driver ζητάει executors από τον τρέχων cluster manager, δεν μαθαίνει αν είναι Standalone, YARN ή Mesos (οι YARN και Mesos μπορεί να δρομολογούν ταυτόχρονα άλλου τύπου εφαρμογές). Ο cluster manager είναι υπεύθυνος για την ανάθεση πόρων σε κάθε executor, ο οποίος αντιστοιχίζεται σε πραγματικούς πόρους, δηλαδή πυρήνες CPU, μνήμη κ.ά. Στη συνέχεια, ο cluster manager ξεκινάει τους Spark executors στο cluster, και ο driver με τη σειρά του χωρίζει την εφαρμογή σε μικρότερα κομμάτια προς εκτέλεση (tasks) και τα αναθέτει στους executors, αντιστοιχίζοντας ένα task σε μία διεργασία (process) ενός worker. Όταν η εκτέλεση του Spark γίνεται σε cluster mode, ο master κόμβος διαλέγει (δικτυακή επικοινωνία) έναν worker κόμβο με αρκετούς διαθέσιμους πόρους για να τρέξει τον driver ως διεργασία. Ο driver χρειάζεται τουλάχιστον έναν πυρήνα και λαμβάνει προεπιλεγμένα 1 GB μνήμης. Στο παρόν mode, η υποβολή της εφαρμογής σημαίνει και τη λήξη της client process (διεργασία πελάτη), ενώ θεωρείται ότι προσφέρει ευκολότερη ανάθεση πόρων αφήνοντας τον master να αποφασίσει. Επίσης, διαθέτει συγκεντρωμένη παρακολούθηση του driver μαζί με τους workers, στο Web UI. Συμφέρει σε περίπτωση που το cluster είναι απομακρυνσμένο από τον χρήστη, καθώς εξαλείφει την καθυστέρηση μεταφοράς δεδομένων μεταξύ του driver και του cluster. Από την άλλη, στο client mode, ο driver εκτελείται σε έναν υπολογιστή εκτός του cluster, κατά την υποβολή μιας εφαρμογής. Πλεονέκτημα αποτελεί ότι οι workers είναι εξ ολοκλήρου διαθέσιμοι για την εκτέλεση tasks, εφόσον ο driver απουσιάζει. Ακόμη, σε περίπτωση διαχείρισης του cluster απομακρυνσμένα, είναι προτιμότερο να γίνει μια RPC κλήση στον (απομακρυνσμένο) driver ώστε να εκτελεί τις ενέργειές

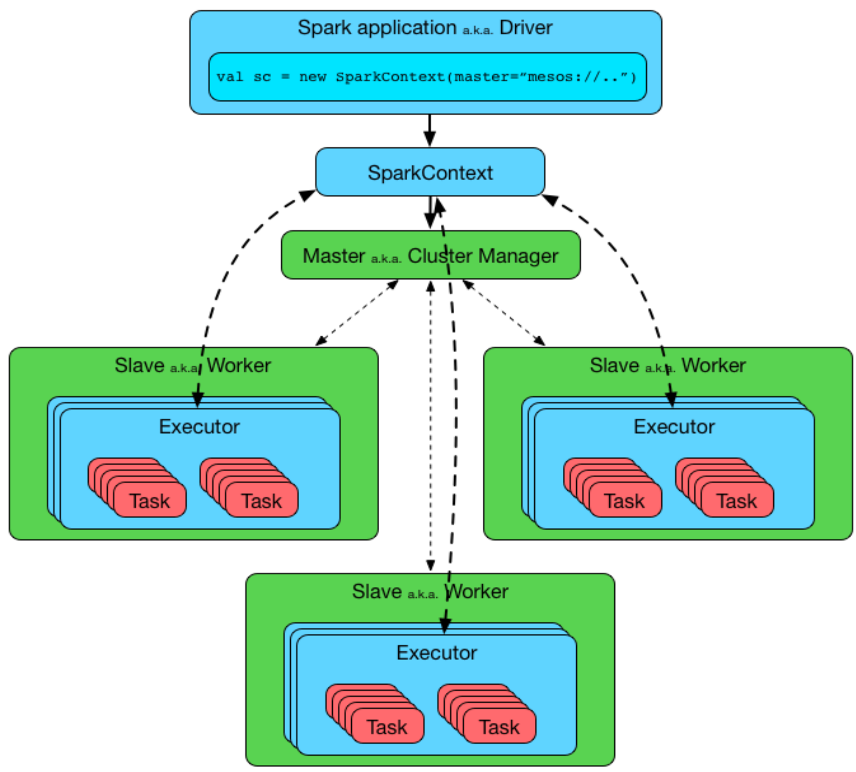
του κοντά στους worker κόμβους. Στη συνέχεια, βλέπουμε δύο υποβολές εφαρμογής στο Spark, με Hadoop YARN ως cluster manager, διακρίνοντας τους πόρους που χρησιμοποιεί ο driver ή πρόγραμμα οδήγησης ή SparkContext:



spark-submit MYJAR --master yarn-cluster –class MYCLASS spark-submit MYJAR --master yarn-client –class MYCLASS

Εικόνα 23: Apache Spark με Hadoop YARN διαχειριστή συμπλέγματος. Cluster Mode με το SparkContext μέσα στο σύμπλεγμα, και Client Mode με το SparkContext έξω από το σύμπλεγμα.

Κάθε εφαρμογή που υποβάλλεται στο Spark, λαμβάνει τους δικούς της executors που μένουν μέχρι το πέρας της εκτέλεσής της, αναλαμβάνοντας tasks και τρέχοντάς τα σε πολλαπλά threads. Συνεπώς, υπάρχει το πλεονέκτημα απομόνωσης των εφαρμογών και σε επίπεδο scheduling (κάθε scheduler δρομολογεί τα δικά του tasks), αλλά και executor (tasks από διαφορικές εφαρμογές βρίσκονται σε διαφορετικό JVM). Αυτό από τη άλλη σημαίνει ότι για την κοινοποίηση δεδομένων μεταξύ των εφαρμογών είναι απαραίτητο να παρεμβληθεί εξωτερικό αποθηκευτικό σύστημα[22]. Παρακάτω διακρίνουμε αναλυτικά τη δικτυακή επικοινωνία μεταξύ των driver, master, worker, executors και cluster manager:



Εικόνα 24: Αρχιτεκτονική Cluster: Επικοινωνία μεταξύ εφαρμογής, driver, SparkContext, Cluster Manager, Workers, Executors.

Έπειτα, βλέπουμε έναν πίνακα με τη βασική ορολογία ενός cluster:

Application	*Πρόγραμμα ή εφαρμογή χρήστη για Spark.* *Αποτελείται από τον driver και τους executors στο cluster.*
Application jar	*Ένα αρχείο JAR που περιέχει την εφαρμογή του χρήστη. Ανάλογα με την εφαρμογή δημιουργείται ένα τεράστιο JAR που περιέχει επιπλέον τις εξαρτήσεις βιβλιοθηκών. Περιττές οι Hadoop και Spark βιβλιοθήκες, καθώς προστίθενται κατά την εκτέλεση.*
Driver program	*Η διεργασία που τρέχει την main() συνάρτηση της εφαρμογής και δημιουργεί το SparkContext.*
Cluster manager	*Μια εξωτερική διαχειριστική υπηρεσία για την ανάθεση πόρων στο σύμπλεγμα (cluster). Π.χ. Standalone Scheduler, Hadoop YARN ή Apache Mesos.*
Deploy mode	*Ξεχωρίζεται από την τοποθεσία που τρέχει η διεργασία του driver.* *Σε "cluster" mode, τρέχει σε κάποιον worker μέσα στο cluster.* *Σε "client" mode, σε κάποιον υπολογιστή έξω από το cluster.*
Worker node	*Κάθε κόμβος που μπορεί να τρέξει κώδικα της εφαρμογής.*
Executor	*Μια διεργασία της εφαρμογής που εκκινεί σε έναν worker κόμβο και τρέχει tasks και κρατάει τα δεδομένα στη μνήμη ή στο δίσκο μεταξύ των άλλων executors.* *Κάθε εφαρμογή έχει τους δικούς της executors.*

Task	Μια μονάδα εργασίας που θα σταλεί σε έναν executor. (Ο driver χωρίζει την εφαρμογή σε μικρότερα κομμάτια, τα tasks.)
Job	Ένας παράλληλος υπολογισμός αποτελούμενος από πολλαπλά tasks που εκκινεί λόγω κάποιας δράσης (Spark action, π.χ. `save`, `collect`). Παρατηρείται σαν όρος στα logs του driver.
Stage	Κάθε Job διαιρείται σε μικρότερα κομμάτια, τα stages, που είναι αλληλοεξαρτώμενα (σαν την εξάρτηση του map και του reduce στο μοντέλο του MapReduce). Παρατηρείται σαν όρος στα logs του driver.

Πίνακας 1: Βασικές έννοιες ενός Cluster.

Επίβλεψη (Monitoring). Κάθε εφαρμογή του Spark διαθέτει ένα Web UI για monitoring. Το Web UI περιβάλλον ξεκινάει το SparkContext της εφαρμογής, ενώ εμφανίζονται πληροφορίες σχετικές με τα tasks, τα jobs, τα stages, τους executors και πόσα tasks ανέλαβε ο καθένας, τα RDDs, τη μνήμη που χρησιμοποιεί η εφαρμογή ή που χρησιμοποίησε, εάν αυτή ολοκληρώθηκε, καθώς και τον αποθηκευτικό χώρο. Ακόμη, δίνει λεπτομέρειες για την δομή του cluster, πόσους workers έχει, πόσους πυρήνες και πόση μνήμη έχει ο κάθε worker και πόσα το cluster συνολικά.

URL: spark://master:7077
REST URL: spark://master:6066 *(cluster mode)*
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 2.0 GB Total, 0.0 B Used
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20171111142711-147.102.19.152-37643	147.102.19.152:37643	ALIVE	1 (0 Used)	1024.0 MB (0.0 B Used)
worker-20171111142711-147.102.19.199-46881	147.102.19.199:46881	ALIVE	1 (0 Used)	1024.0 MB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20171111161613-0000	Recommender	2	1024.0 MB	2017/11/11 16:16:13	kapsoulis	FINISHED	24 s

Εικόνα 25: Παράδειγμα γραφικού περιβάλλοντος Standalone Scheduler με 2 worker nodes, 2 cores (συνήθως 1 core/worker), 1 ολοκληρωμένη εφαρμογή και συνολική μνήμη στο cluster 2 GB.

Στο επόμενο κεφάλαιο περιγράφουμε το πρόβλημα της παρούσας διπλωματικής εργασίας και αναλύουμε την αρχιτεκτονική και τα κομμάτια που την απαρτίζουν.

3

Εισαγωγή

Είδαμε ότι στη σύγχρονη εποχή Τεχνολογίας της Πληροφορίας η χρήση των Big Data, Cloud Computing και Anything-as-a-Service παίζει βασικό ρόλο. Η εξέλιξη αυτή έχει ανοίξει τις πόρτες σε κάθε χρήστη, που είναι συνδρομητής, να απολαμβάνει υπηρεσίες μέσω Internet, χωρίς δικό του εξοπλισμό. Έτσι και στην παρούσα διπλωματική εργασία παρουσιάζουμε μια υπηρεσία που προσφέρει στους πελάτες της διαφορετικά προϊόντα ανάλογα με τις επιλογές τους. Τα δομικά στοιχεία και η αρχιτεκτονική του όλου συστήματος αναλύονται σε αυτό Κεφάλαιο.

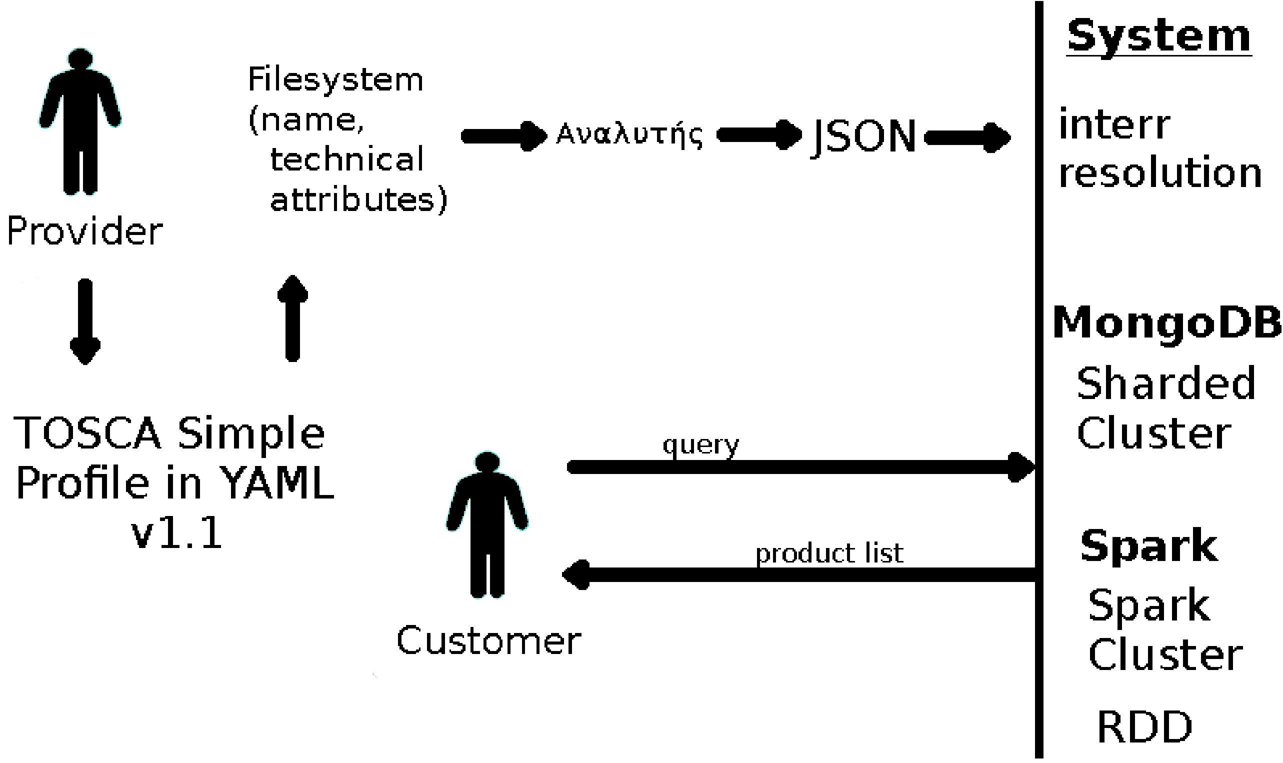
Top-bottom αρχιτεκτονική του συστήματος

Κατ' αρχάς, στα πλαίσια διαχωρισμού των χρηστών που αλληλεπιδρούν με το σύστημά μας, ορίσαμε δύο ειδών φυσικά πρόσωπα: τους παρόχους και τους πελάτες. Οι μεν συμβάλλουν στο να προσφέρουν τα προϊόντα τους, ενώ οι δε στο να καταναλώνουν τα προϊόντα των παρόχων. Η υπηρεσία μας έχει το ρόλο αυτού που επικοινωνεί και με τους δύο, οργανώνοντας τα προϊόντα των παρόχων, και ξεχωρίζοντας και προσφέροντας στους πελάτες τα φθηνότερα προϊόντα που σχετίζονται περισσότερο με αυτό που επιθυμούν.

Τα προϊόντα εδώ ορίζονται ως αποθηκευτικός χώρος (file system storage) συγκεκριμένων προδιαγραφών και χαρακτηριστικών. Η ουσία του συστήματός μας βρίσκεται στην διαφορετικότητα αυτών των χαρακτηριστικών και στις σχέσεις αλληλεπίδρασης μεταξύ τους (αλληλοσυσχετίσεις – interrelationships).

Με μια ματιά, η εκ των άνω προς τα κάτω προσέγγιση του συστήματος έχει ως εξής: ο χρήστης της υπηρεσίας επιλέγει χαρακτηριστικά και προδιαγραφές που επιθυμεί να έχει ο παρεχόμενος

αποθηκευτικός χώρος, η υπηρεσία μας αναλύει το αίτημά του και τού επιστρέφει τα κατάλληλα προϊόντα. Ανά πάσα στιγμή μπορούν να προστεθούν στο σύστημα νέα προϊόντα παρόχων.



Εικόνα 26: Βασικά components της αρχιτεκτονικής του συστήματος

Πάροχος μπορεί να είναι οποιοσδήποτε οργανισμός μπορεί να παρέχει τέτοιου είδους προϊόντα (κυρίως για Cloud Providers, όπως Microsoft, Google, Amazon, Fuga, Platform9, Cloudsigma, CloudVPS κ.ά.). Απαραίτητη προϋπόθεση συμμετοχής του στο σύστημά μας είναι να διαθέτει κατάλληλο εξοπλισμό και υποδομές.

Πελάτης μπορεί να είναι οποιοδήποτε άτομο ή οργανισμός επιθυμεί αποθηκευτικό χώρο (file storage) με συγκεκριμένα χαρακτηριστικά. Τα χαρακτηριστικά αυτά είναι διαφορετικά ανάλογα με τη χρήση για την οποία προορίζονται. Για αυτό το λόγο και ο κάθε πελάτης εκφράζει την επιθυμία του με λέξεις-κλειδιά, και στη συνέχεια το σύστημά μας, αναλύει την πληροφορία που έχει για τα προϊόντα και τού επιστρέφει μια λίστα από αυτά που τον εξυπηρετούν καλύτερα.

Τα χαρακτηριστικά ενός αποθηκευτικού χώρου μπορούν να ανήκουν σε διαφορετικές κατηγορίες και έτσι η πιθανή κατηγοριοποίηση αυτών ποικίλει. Είναι γεγονός ότι εμπίπτουν στην οντότητα μεταδεδομένων (metadata), καθώς πρόκειται για τρόπους και μεθόδους διαφορετικού τύπου αποθήκευσης δεδομένων. Δηλαδή αποτελούν δεδομένα για δεδομένα. Στην παρούσα διπλωματική εργασία οι κατηγοριοποιήσεις μεταδεδομένων που εξετάσαμε είναι τρεις, ενώ εμείς

για τα χαρακτηριστικά των file systems επιλέξαμε ένα συγκεκριμένο διαχωρισμό: technical και business. Οι τρεις κατηγοριοποιήσεις αναλύονται περαιτέρω στο Παράρτημα 1.

Ανάλυση file system attributes

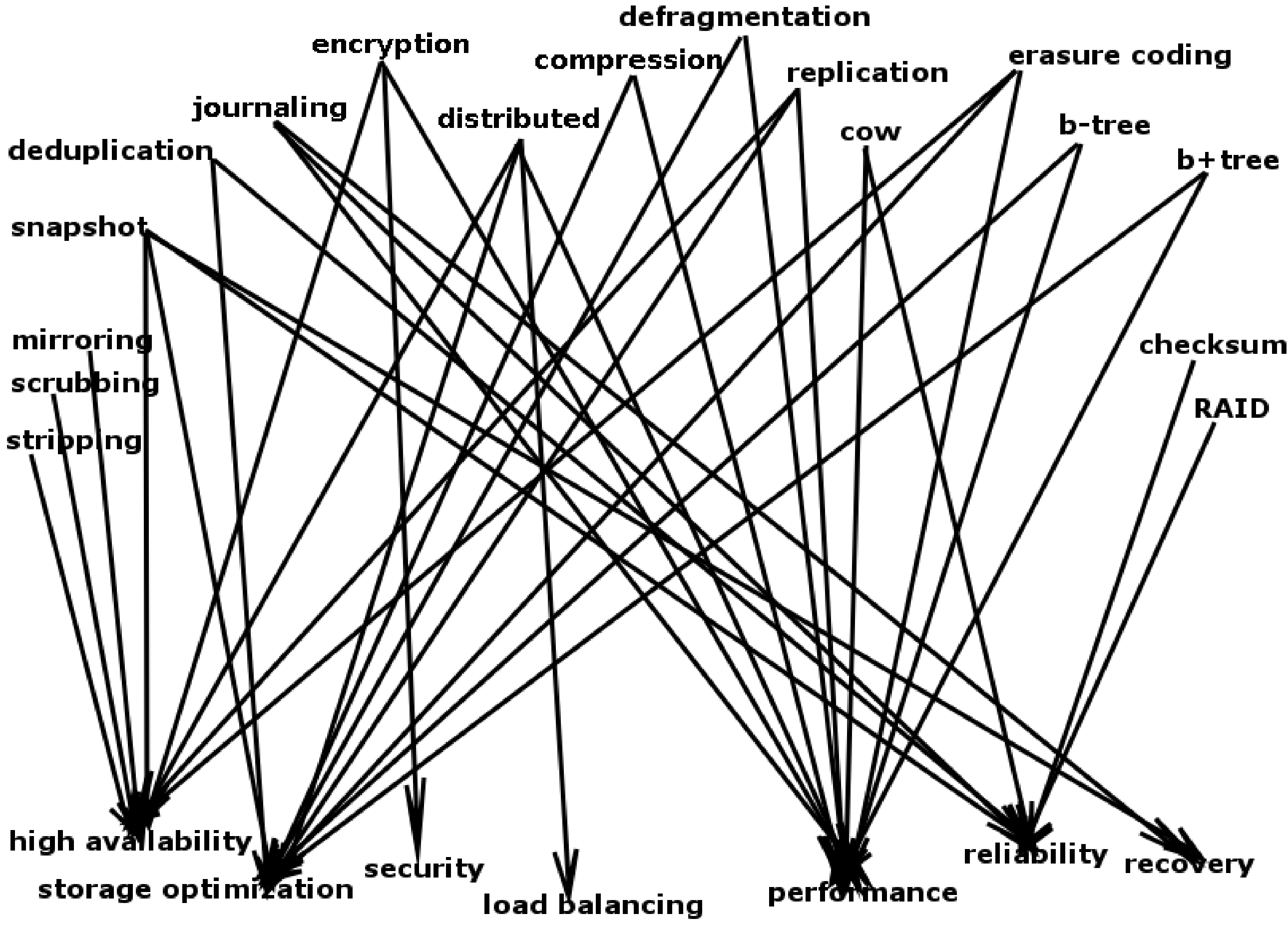
Στην παρούσα εργασία, τα χαρακτηριστικά (attributes) ενός αποθηκευτικού χώρου (file system storage) χωρίζονται, όπως είδαμε, σε technical και business. Technical χαρακτηριστικά είναι όσα έχουν να κάνουν με την εσωτερική δομή ενός file system. Τέτοια είναι τα journaling, encryption, distributed, compression, erasure coding, replication, cow, b-tree, b+tree, defragmentation, deduplication, snapshot, checksum, RAID, mirroring, scrubbing, stripping κλπ. Πρόκειται για χαρακτηριστικά που εξαρτώνται συνήθως από αυτά καθ' αυτά και όχι από άλλους παράγοντες. Κάποιο προϊόν (file system storage) μπορεί να υποστηρίζει ή να μην υποστηρίζει ένα technical attribute. Κάποιες φορές γίνονται και τα δύο. Ας δούμε παρακάτω ένα παράδειγμα.

Το Btrfs (B-tree file system)[6] έχει δημιουργηθεί για να υποστηρίζει copy-on-write (cow), αλλά ταυτόχρονα διαθέτει την “nodatacow” flag. Αυτή η μεταβλητή όταν τεθεί δεν επιτρέπει copy-on-write για νέα αρχεία. Στην πραγματικότητα, μιλώντας για χαρακτηριστικά file systems, βλέπουμε ότι η συγκεκριμένη επιλογή έχει μεγάλη σημασία. Κι αυτό γιατί αποφασίζουμε εάν θέλουμε ή όχι καλύτερη απόδοση (performance) του συστήματος έναντι αξιόπιστης αποθήκευσης αρχείων στο δίσκο (reliability) σε περίπτωση βλάβης (system failure – λ.χ. διακοπή ρεύματος ή system crash). Η παρουσία του technical χαρακτηριστικού "cow" βελτιώνει κάποιο business χαρακτηριστικό, ενώ η απουσία του βελτιώνει ένα διαφορετικό business χαρακτηριστικό. Άρα βλέπουμε ότι ένας πάροχος μπορεί να διαθέτει ταυτόχρονα:

- Btrfs, με copy-on-write (cow) → καλύτερη αξιοπιστία (reliability)
- Btrfs, με nodatacow enabled → καλύτερη απόδοση (performance)

Από την άλλη, τα business χαρακτηριστικά (attributes) ενός αποθηκευτικού χώρου (file system storage), όπως βλέπουμε, εξαρτώνται άμεσα από τα technical και μπορεί να είναι performance, reliability, recovery, security, high availability, storage optimization, load balancing, κλπ. Η ύπαρξή τους προκύπτει άμεσα από την ύπαρξη ενός τουλάχιστον technical χαρακτηριστικού. Ουσιαστικά τα τεχνικά χαρακτηριστικά είναι ο λόγος και η εξήγηση των business χαρακτηριστικών. Για παράδειγμα, το HDFS έχει υψηλή διαθεσιμότητα (high availability), επειδή είναι κατανεμημένο (distributed) σύστημα αρχείων. Όταν κάποιο file system έχει υψηλή διαθεσιμότητα, αναρωτιόμαστε από πού προέρχεται αυτή. Σε αυτό το πλαίσιο αίτιου-αποτελέσματος, βασίζουμε τη λειτουργικότητα του συστήματός μας, δίνοντας βάση στα τεχνικά χαρακτηριστικά (technical) και λαμβάνοντας υπ' όψιν τα business.

Επομένως, για το σύστημά μας, το κάθε technical χαρακτηριστικό θεωρείται αυτόνομο, ενώ το κάθε business θεωρείται εξαρτώμενο από κάποιο technical. Όταν ένα προϊόν παρόχου προστίθεται στο σύστημα, έχει όνομα, όνομα παρόχου, είδος υπηρεσίας (εδώ, προεπιλεγμένη η τιμή "FileSystem as a Service") και λίστα τεχνικών χαρακτηριστικών (technical attributes) δοσμένα από τον πάροχο. Στην πορεία, ανάλογα με το όνομα και τα τεχνικά χαρακτηριστικά (technical) του κάθε προϊόντος δημιουργούνται τα χρήσιμα πεδία "business" και "price" που είναι αντίστοιχα τα business χαρακτηριστικά και η τιμή του. Αυτό συμβαίνει με την χρήση ζευγαριών technical και business σαν αίτιο-αποτέλεσμα. Όπως είδαμε στο παραπάνω παράδειγμα, η ύπαρξη ή μη copy-on-write (technical) δίνει και διαφορετικά business χαρακτηριστικά. Αυτές τις σχέσεις αιτίου και αποτελέσματος ονομάζουμε αλληλοσυσχετίσεις (interrelationships) και αποτελούν τον βασικό πυλώνα της ανάλυσης. Κάποια παραδείγματα βλέπουμε παρακάτω:



Εικόνα 27: Interrelationships – Αλληλοσυσχετίσεις. Οι συντελεστές προσθαφαίρεσης επιλέγονται ώστε κάθε technical να επηρεάζει θετικά ή αρνητικά ένα business χαρακτηριστικό.

Συνεχίζοντας σχετικά με την εισαγωγή προϊόντων στο σύστημα, είδαμε την δημιουργία των business χαρακτηριστικών και τώρα θα δούμε πώς με παρόμοιο τρόπο καθορίζεται η τιμή ενός προϊόντος. Από το σύστημα, ορίζεται μία συγκεκριμένη αξία (price) για κάθε τεχνικό χαρακτηριστικό ανεξάρτητα με το προϊόν, και μία για κάθε file system μόνο και μόνο από το όνομά του. Η συνολική τιμή στην οποία αγοράζεται από τον πελάτη είναι το άθροισμα της αξίας του ονόματος και της αξίας κάθε τεχνικού χαρακτηριστικού. Για παράδειγμα, το Btrfs παραπάνω αξίζει από μόνο του 80.0 νομισματικές μονάδες, λόγω του ονόματός του το οποίο αποτιμάται από το σύστημα. Στην προκειμένη περίπτωση, έχει επιπλέον το τεχνικό χαρακτηριστικό copy-on-write, που αποτιμάται επίσης από το σύστημα στα 50 νομισματικές μονάδες, άρα η συνολική τιμή που το αγοράζει ο πελάτης προκύπτει από το άθροισμα της τιμής του ονόματος και των τιμών των τεχνικών χαρακτηριστικών (στο παράδειγμα μόνο το copy-on-write), 130 νομισματικές μονάδες.

Μέχρι στιγμής είδαμε τη γενική εικόνα της εκ των άνω προς τα κάτω αρχιτεκτονικής του συστήματος. Μπαίνοντας πλέον σε λεπτομερέστερα κομμάτια αυτής της αρχιτεκτονικής, θα δούμε τα δομικά στοιχεία της, πράγμα που θα μας βοηθήσει να καταλάβουμε βασικά κομμάτια του κώδικα το επόμενο κεφάλαιο.

Χρήση του προτύπου TOSCA Simple Profile in YAMLv1.1

Ας ξεκινήσουμε από το πώς ο πάροχος προσφέρει τις σχετικές πληροφορίες για τα προϊόντα του στο σύστημα. Γίνεται χρήση της προδιαγραφής του οργανισμού OASIS[8], TOSCA (Topology and Orchestration Specification for Cloud Applications) Simple Profile in YAML Version 1.1[9]. Η προδιαγραφή αυτή αποτελεί μια πρότυπη γλώσσα περιγραφής τοπολογίας υπηρεσιών ιστού βασιζόμενων σε υπολογιστικό νέφος (Cloud Computing). Περιέχει προδιαγραφές για την περιγραφή λειτουργιών που δημιουργούν ή τροποποιούν υπηρεσίες ιστού (web services). Το μοντέλο TOSCA, όπως λεγεται, χρησιμοποιεί την έννοια πρότυπα υπηρεσίας (service templates) για να περιγράψει φόρτο εργασίας νέφους (cloud workloads) με τη βοήθεια πρότυπων τοπολογιών (κόμβοι) και πρότυπων σχέσεων μεταξύ αυτών. Στο Παράρτημα 1, δίνουμε ένα απλό και εισαγωγικό παράδειγμα και αναλύουμε περαιτέρω την προδιαγραφή.

Στην παρούσα εργασία χρειάστηκε να δημιουργήσουμε έναν τύπο κόμβου (node type) σύμφωνα με την προδιαγραφή, και να υποχρεώνουμε τον κάθε πάροχο να περιγράφει με αυτόν τον τρόπο το κάθε προϊόν του. Ο τύπος αυτός επεκτείνει τον ήδη υπάρχων τύπο κόμβου (node type) "tosca.nodes.BlockStorage", ο οποίος αντιπροσωπεύει ίδιου μεγέθους θέσεις μνήμης στο δίσκο (blocks) που μπορούν να δημιουργήσουν αποθηκευτικό χώρο. Η δημιουργία νέου τύπου κόμβου σύμφωνα με το TOSCA Simple Profile YAMLv1.1 είναι αναγκαία, καθώς προσθέτουμε πεδία στο αρχείο τύπου YAML, που είναι συγκεκριμένα για το σύστημά μας και τον τρόπο λειτουργίας του και τα οποία δεν υπήρχαν προηγουμένως. Ακόμα, ο πάροχος δεν επιβαρύνεται με

φόρτο εργασίας αφού το μόνο που έχει να κάνει είναι να δηλώσει το όνομα τού κάθε προϊόντος (product_name) μαζί με τα τεχνικά του χαρακτηριστικά (attributes) [βλ. παρακάτω]. Ας σημειώσουμε ότι το όνομα του παρόχου (cloud provider name) αποθηκεύεται ανάλογα με τα αρχεία που δίνει ο ίδιος και δεν χρειάζεται να υπάρχει μέσα στο αρχείο τύπου YAML. Παρακάτω, βλέπουμε αναλυτικότερα τον νέο τύπο κόμβου (node type) που χρησιμοποιήσαμε:

```
node_types:
 FilesystemAsAService:
  derived_from: tosca.nodes.BlockStorage
  properties:
   product_name:
    type: string
    description: Product's official name
   attributes:
    type: list
    entry_schema:
     description: Filesystem attribute's name (simple string entry)
     type: string
```

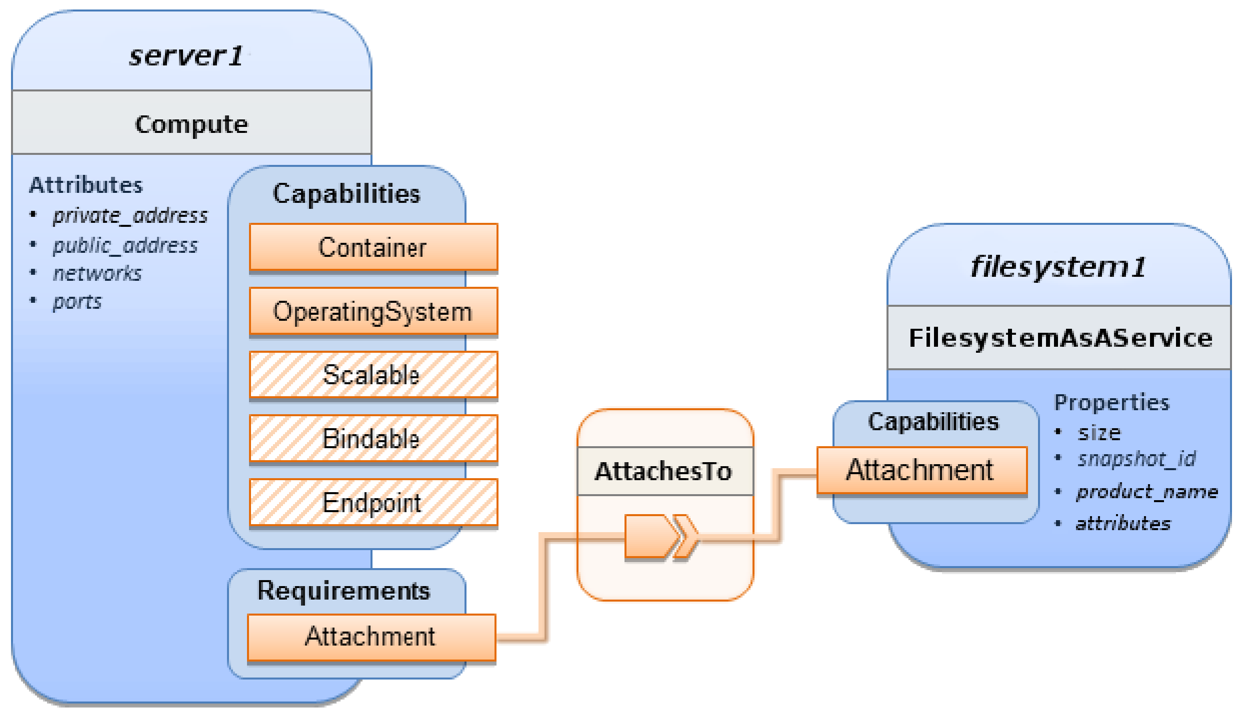
Ο τύπος "tosca.nodes.BlockStorage" έχει από μόνος του τα πεδία "size" και "snapshot_id" που αντιστοιχούν στο μέγεθος και κάποιο υπάρχον snapshot του filesystem. Η τελευταία τιμή δεν είναι συγκεκριμένη, γι' αυτό όπως θα δούμε παρακάτω δίνεται από το πεδίο "inputs" κατά την ανάπτυξη της εφαρμογής.

Οι τροποποιήσεις και επεκτάσεις στο μοντέλο TOSCA δεν σταματούν βέβαια στον νέο τύπο κόμβου (node type) και στην τροποποίηση του πεδίου "node_types". Το επόμενο πεδίο του YAML αρχείου που κατασκευάζουμε είναι το "topology_template" και περιέχει τα "inputs" και "node_templates" πεδία που θα μας απασχολήσουν. Ας σημειώσουμε ότι παρακάτω θα δούμε ολόκληρο το αρχείο YAML που περιγράφει όλα τα απαραίτητα για την εργασία πεδία και θα διακρίνουμε καλύτερα πώς δένουν μεταξύ τους. Προς το παρόν, στο επόμενο υποχρεωτικό από την προδιαγραφή TOSCA πεδίο, το "topology_template", έχουμε με τη σειρά τα πεδία "inputs" και "node_templates" όπως ήδη αναφέραμε. Στο τελευταίο, περιγράφονται οι κόμβοι (nodes) και οι δυνατότητες και σχέσεις (relationships) μεταξύ τους (βλ. Παράρτημα 1 για nodes και relationships), ενώ στο πρώτο ορίζονται μεταβλητές που αφορούν λεπτομέρειες του προηγούμενου και θα εισαχθούν κατά την ανάπτυξη της εκάστοτε εφαρμογής (βλ. Παράρτημα 1 για "inputs"). Σχετικά με τα "node_templates", ορίζουμε έναν κόμβο τύπου "tosca.nodes.Compute" με όνομα "server1" και σε αυτόν συνδέουμε (capabilities → requirements → - local_storage → node: filesystem1) μέσω της σχέσης "AttachesTo" (capabilities → requirements → - local_storage → relationship → type: FilesystemAsAService) έναν κόμβο τύπου FilesystemAsAService, (προϊόν, file system storage) με όνομα "filesystem1".

```
node_templates:
 server1:
  type: tosca.nodes.Compute
  capabilities:
    host:
     properties:
     # omitted, no interest here
    os:
     properties:
     # omitted, no interest here
    requirements:
     - local_storage:
        node: filesystem1
        relationship:
         type: AttachesTo
         properties:
          location: { get_input: storage_location1 }
    # storage_location1: Block storage mount point (filesystem path) for filesystem1
    # This maps the local requirement name 'local_storage' to the
    # target node's capability name 'attachment'

 filesystem1:
  type: FilesystemAsAService
  properties:
   size: 10 GB
   snapshot_id: { get_input: storage_snapshot_id1 }
   product_name: 'ReFS'
   attributes: ['replication', 'snapshot']
```

Τέλος, θα δούμε το συνολικό YAML αρχείο που ακολουθεί την προδιαγραφή TOSCA Simple Profile YAML version 1.1, τονίζοντας τις λέξεις-κλειδιά που αλληλεπιδρούν μεταξύ τους αλλά και το πεδίο "inputs". Το τελευταίο, περιέχει μεταβλητές που δεν είναι σταθερές για κάθε φορά που γίνεται ανάπτυξη των συγκεκριμένων εφαρμογών και υπηρεσιών, αλλά λαμβάνουν την τιμή τους εκείνη τη στιγμή. Παραδείγματα τέτοιων μεταβλητών είναι κωδικοί, αριθμός θύρας, IP κ.ά. Στην περίπτωσή μας είναι το "location", το σημείο που θα γίνει προσβάσιμο το "filesystem1" κατά την ανάπτυξη και το "snapshot_id" που εξηγήσαμε παραπάνω.



Εικόνα 28: Σύνδεση ενός FilesystemAsAService, επέκταση του TOSCA BlockStorage κόμβου, με έναν TOSCA Compute κόμβο, με χρήση της TOSCA σχέσης AttachesTo.

```
# Provider's name: different YAML files come from different providers.
tosca_definitions_version: tosca_simple_yaml_1_0

description: A provider's TOSCA Simple Profile YAML. We are interested only in information about filesystem/storage attributes. The YAML may contain one or more 'Products' (i.e. filesystem/storage as a service) - inputed as class and converted to JSON inside our application.

# node_types [optional], it lists the Node Types that provide the reusable type definitions for software components that Node Templates can be based upon.
node_types:
 FilesystemAsAService:
  derived_from: tosca.nodes.BlockStorage
  properties:
   product_name:
    type: string
    description: Product's official name
   attributes:
    type: list
    entry_schema:
     description: >
Filesystem attribute's name (simple string entry)
     type: string
# omitted, no more interest here

topology_template:
```

```yaml
inputs:
 storage_snapshot_id1:
  type: string
  description: >
Optional identifier for an existing
snapshot to use when creating storage.
 storage_location1:
  type: string
  description: >
Block storage mount point
(filesystem path) for filesystem1.
 # omitted, no interest here

 # node_templates, it lists the Node Templates
 # that describe the (software) components
 # that are used to compose cloud applications.
 node_templates:
  server1:
   type: tosca.nodes.Compute
   capabilities:
     host:
      properties:
      # omitted, no interest here
     os:
      properties:
      # omitted, no interest here
     requirements:
      - local_storage:
         node: filesystem1
         relationship:
          type: AttachesTo
          properties:
           location: { get_input: storage_location1 }

  filesystem1:
   type: FilesystemAsAService
   properties:
    size: 10 GB
    snapshot_id: { get_input: storage_snapshot_id1 }
    product_name: 'ReFS'
    attributes: ['replication', 'snapshot']
```

Εφόσον είδαμε τον τρόπο που λαμβάνονται τα δεδομένα, δηλαδή μέσω της μορφής YAML version 1.1 αρχείου, θα συνεχίσουμε με την εισαγωγή των προϊόντων στο σύστημα και πώς επιτεύχθηκε αυτό, εφόσον προηγουμένως τονίσουμε ένα σημαντικό σημείο.

Χρήση της αναπαράστασης JSON

Η μορφή YAML δεν είναι κατάλληλη για επεξεργασία και ανάλυση μεγάλων δεδομένων (big data), καθώς εμφανίζεται αρκετά πιο αργή στη σειροποίηση και αποσειριοποίηση από άλλες γλώσσες αναπαράστασης αρχείων[11]. Συγκρίνοντας με την JSON, βλέπουμε ότι και οι δύο στοχεύουν να είναι ευανάγνωστες από τον άνθρωπο αναπαραστάσεις δεδομένων. Εντούτοις, έχουν διαφορετικές προτεραιότητες. Πρωταρχικός σχεδιαστικός στόχος της JSON είναι η απλότητα και η καθολικότητα, γι' αυτό και η παραγωγή και συντακτική ανάλυση ενός JSON είναι τετριμμένη, με κόστος βέβαια στην ευκολία ανάγνωσης. Επιπλέον, το μοντέλο αναπαράστασης πληροφορίας που χρησιμοποιεί, ακολουθεί τη λογική του ελάχιστου κοινού πολλαπλασίου, εξασφαλίζοντας με αυτή τη λογική ότι κάθε JSON αρχείο θα μπορεί εύκολα να επεξεργαστεί από νέα προγραμματιστικά περιβάλλοντα. Από την άλλη, ο πρωταρχικός σχεδιαστικός στόχος της YAML έγκειται στην ευκολία ανάγνωσης και τη δυνατότητα σειριοποίησης πιο αυθαίρετων δομών δεδομένων. Συνεπώς, σε αντίθεση με τη JSON, η YAML προσφέρει πολύ υψηλή ευκολία ανάγνωσης από τον άνθρωπο, με κόστος βέβαια στην παραγωγή και συντακτική ανάλυσης αρχείου. Ακόμη, το μοντέλο αναπαράστασης της πληροφορίας απέχει αρκετά από τη λογική ελάχιστου κοινού πολλαπλασίου, όπως στη JSON, και απαιτείται πολυπλοκότερη επεξεργασία μεταφοράς μεταξύ προγραμματιστικών περιβάλλοντων. Είναι γεγονός ότι η YAML αποτελεί φυσικό υπερσύνολο της JSON[14], προσφέροντας βελτιωμένη αναγνωσιμότητα αλλά και πιο ολοκληρωμένο μοντέλο αναπαράστασης πληροφορίας. Με την πάροδο των χρόνων χρήσης τήςYAML από τη στιγμή δημιουργίας της, ο σκοπός της άλλαξε. Από markup γλώσσα, δηλαδή γλώσσα για βέλτιστη αναπαράσταση της πληροφορίας, έγινε data-oriented γλώσσα, δίνοντας βάση στην περιεχόμενη πληροφορία. Γι' αυτό άλλαξε και το όνομά της από Yet Another Markup Language στο αναδρομικό YAML Ain't Markup Language[12].

Για τους παραπάνω λόγους, επιλέξαμε να χρησιμοποιήσουμε στο σύστημά μας JSON, πιο γρήγορη σε σειροποίηση και αποσειριοποίηση από τη YAML[11]. Η JSON στις μέρες μας αποτελεί μία από τις βασικές μορφές ανταλλαγής δεδομένων στο σύγχρονο Internet. Υποστηρίζει όλες τις βασικές μορφές δεδομένων, δηλαδή αριθμούς, συμβολοσειρές, boolean, πίνακες και hashes. Σχετικά με τη μετατροπή αρχείου από YAML σε JSON, κατασκευάσαμε έναν απλό συντακτικό αναλυτή. Χρησιμοποιείται για να διαβάζει το YAML αρχείο του παρόχου και να βγάζει ως έξοδο την ίδια πληροφορία σε JSON μορφή, κάνοντας απλή αντιστοίχιση των τιμών του κάθε file system ενός YAML αρχείου σε JSON σύνταξη. Ο συντακτικός αναλυτής είναι αποδοτικός σε χρόνο καθώς διαβάζει συγκεκριμένες γραμμές του YAML αρχείου του cloud provider, όσες δηλαδή είναι σχετικές με Filesystem as a Service, που είναι το προϊόν που μας ενδιαφέρει στην παρούσα εργασία.

Για παράδειγμα, αν το παρακάτω κομμάτι βρίσκεται στο YAML αρχείο της Amazon:

```
...
```

```
  filesystem17:
   type: FilesystemAsAService
   properties:
    size: 10 GB
    snapshot_id: { get_input: storage_snapshot_id17 }
    product_name: 'HDFS'
    attributes: ['distributed', 'snapshot', 'erasure_coding']
...
```

τότε, με τη βοήθεια του αναλυτή μας, η μετατροπή του σε JSON document δίνει:

```
...,
{ "name" : "HDFS",
  "service_type" : "FilesystemAsAService",
  "provider" : "Amazon",
  "price" : 930.0,
  "technical" : ["erasure_coding", "snapshot", "distributed"],
  "business" : ["high_availability", "storage_optimization",
"reliability", "recovery", "load_balancing"]
},
...
```

Εδώ, γίνεται ξεκάθαρο, ότι κατά τη μετατροπή αυτή ο συντακτικός αναλυτής μας λαμβάνει υπόψιν του όλα όσα έχουν ειπωθεί στο παρόν κεφάλαιο σχετικά με αλληλοσυσχετίσεις (interrelationships) τεχνικών και business χαρακτηριστικών, αλλά και υπολογισμού της τιμής του προϊόντος με βάση το ονόμα και τα τεχνικά του χαρακτηριστικά (technical).

Στη συνέχεια, εξηγούμε την είσοδο των δεδομένων μορφής αρχείου JSON στη βάση MongoDB, τους λόγους χρήσης της καθώς και την αρχιτεκτονική της στην παρούσα εργασία.

Χρήση της MongoDB

Η MongoDB αποτελεί μια εγγραφο-κεντρική βάση δεδομένων (document-oriented) και χρησιμοποιεί JSON documents για να αποθηκεύει εγγραφές δεδομένων, με τον ίδιο τρόπο που οι σχεσιακές βάσεις δεδομένων χρησιμοποιούν πίνακες και σειρές για την αποθήκευση δεδομένων. Μια βάση δεδομένων χαρακτηρίζεται ως JSON βάση δεδομένων, όταν επιστρέφει αποτελέσματα ερωτημάτων που μπορούν εύκολα να διαβαστούν, με ελάχιστη μετατροπή, απευθείας με τη βοήθεια της γλώσσας JavaScript ή άλλων δημοφιλών γλωσσών προγραμματισμού – μειώνοντας με αυτόν τον τρόπο τον κόπο που πρέπει να καταβάλλει μια εφαρμογή για την επικοινωνία με τη βάση.

BSON

Η MongoDB αντιστοιχεί ένα JSON έγγραφο που εισάγεται σε δυαδική μορφή (binary) εσωτερικά, ονομάζοντάς το BSON document (Binary JSON). Το BSON document επεκτείνει το JSON μοντέλο για να προσφέρει επιπλέον τύπους δεδομένων και πεδία, και το κυριότερο για να είναι αποδοτικότερο για κωδικοποίηση και αποκωδικοποίηση από διαφορετικές γλώσσες προγραμματισμού.

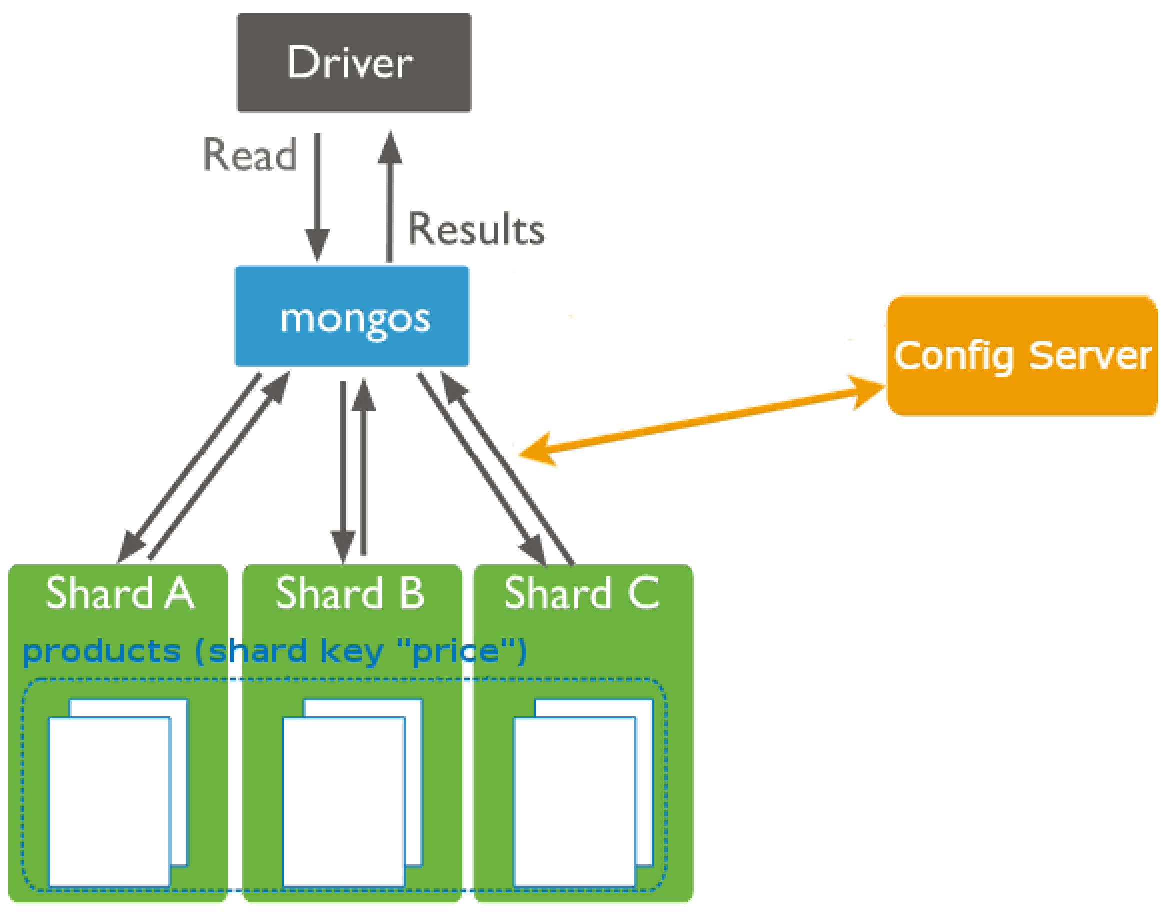
Το BSON document της MongoDB είναι γρήγορο, ελαφρύ και εύκολα διασχίσιμο. Όπως και το JSON, το BSON της MongoDB υποστηρίζει εμφωλευμένα αντικείμενα και πίνακες μέσα σε άλλα αντικείμενα (objects) και πίνακες (arrays). Αυτό σημαίνει ότι η MongoDB μπορεί να επεμβαίνει μέχρι και στα πιο βαθιά εμφωλευμένα κλειδιά BSON αντικειμένων για την απάντηση κάποιου ερωτήματος ή την ανανέωση κάποιας τιμής, τόσο εύκολα όσο στα πρώτου επιπέδου κλειδιά. Συνεπώς, η MongoDB δίνει στους χρήστες την ευκολία και ελαστικότητα ενός JSON document μαζί με την ταχύτητα μιας ελαφριάς binary μορφής.

Sharding

Τα παραπάνω αποτελούν βασικούς λόγους χρήσης της MongoDB στο σύστημά μας, αλλά όχι απαραίτητα τους μοναδικούς. Η MongoDB προσφέρει μία μέθοδο για να διανέμει τα δεδομένα σε πολλαπλά μηχανήματα, το sharding (θρυμματισμός), όπως είδαμε και στο Κεφάλαιο 2. Η MongoDB χρησιμοποιεί το sharding για να υποστηρίζει εφαρμογές με μεγάλα δεδομένα (big data) και με λειτουργίες που απαιτούν υψηλό throughput, εφαρμογές, δηλαδή, που δεν εκτελούνται αποδοτικά ή και καθόλου πάνω σε έναν μόνο server. Μέσω του sharding, η MongoDB πετυχαίνει οριζόντια κλιμάκωση (horizontal scaling), που σημαίνει επέκταση ενός κατανεμημένου συστήματος προσθέτοντας επιπλέον αριθμό μηχανημάτων τα οποία θα αναλαμβάνουν, πλέον, κλάσμα του φόρτου εργασίας. Σημαντικές έννοιες του sharding της MongoDB είναι τα "chunks", δηλαδή τα κομμάτια, ίσου μεγέθους, στα οποία χωρίζει τα δεδομένα, τα "shards", στα οποία προσπαθεί να ισοδιανέμει τα chunks – είναι σωστό να υπάρχει 1 shard ανά μηχάνημα – και το "shard key", που είναι το κλειδί με το οποίο χωρίζει τα δεδομένα. Για κάθε sharded cluster, δηλαδή κατανεμημένο σύστημα MongoDB με sharding, είναι απαραίτητοι οι εξείς μηχανισμοί: τα shards (2 ή περισσότερα), τα mongos (ή query routers, 1 ή περισσότεροι) και οι config servers (1 ή περισσότεροι). Με το sharding ένα κατανεμημένο σύστημα ανταποκρίνεται καλύτερα σε εφαρμογές υψηλού throughput και μεγάλων δεδομένων.

Το sharding της MongoDB που χρησιμοποιήθηκε στην παρούσα εργασία, αναπτύχθηκε πάνω σε 3 εικονικά μηχανήματα. Το ένα από τα τρία μηχανήματα σηκώνει 1 shard, 1 config server και 1 mongos, ενώ τα άλλα 2 μηχανήματα σηκώνουν από 1 shard το καθένα. Ο driver της MongoDB είναι ουσιαστικά μια βιβλιοθήκη που επιτρέπει την επικοινωνία με την ίδια τη βάση μέσω κάποιας

γλώσσας προγραμματισμού. Στην παρούσα εργασία όπως θα δούμε στο επόμενο κεφάλαιο, χρησιμοποιούμε κυρίως τον driver της Scala.



Εικόνα 29: Η συλλογή products κατανέμεται στα 3 shards. Το ένα εικονικό μηχάνημα φιλοξενεί ταυτόχρονα 1 mongos, 1 config και 1 shard, ενώ τα άλλα 2 μηχανήματα από 1 shard.

Για τους παραπάνω λόγους, η MongoDB επιλέχθηκε να χρησιμοποιηθεί στην παρούσα εργασία ως η βάση δεδομένων, όπου σε πρώτη φάση θα αποθηκεύονται τα προϊόντα των παρόχων. Όπως θα δούμε σε λίγο, η δομή της κρίνεται κατάλληλη για ταχύτερη εύρεση απαντήσεων σε queries και για αποδοτικότερη επικοινωνία με το Spark που εκτελεί υπολογισμούς.

Εισαγωγή προϊόντων στο σύστημα

Εισάγουμε στη βάση το παρακάτω JSON που προκύπτει, για παράδειγμα, από το TOSCA Simple Profile YAML της Microsoft:

```
[
 { "name" : "HAMMER",
   "service_type" : "FilesystemAsAService",
   "provider" : "Microsoft",
   "price" : 350.0,
   "technical" : ["snapshot", "b+tree"],
   "business" : ["reliability", "recovery", "high_availability",
"performance"] },

 { "name" : "eCryptFS",
   "service_type" : "FilesystemAsAService",
   "provider" : "Microsoft",
   "price" : 270.0,
   "technical" : ["encryption"],
   "business" : ["security"] },

 {  "name" : "ReFS",
   "service_type" : "FilesystemAsAService",
   "provider" : "Microsoft",
   "price" : 790.0,
   "technical" : ["scrubbing", "checksum", "b+tree", "deduplication"],
   "business" : ["high_availability", "reliability", "performance"] },

{  "name" : "MooseFS",
   "service_type" : "FilesystemAsAService",
   "provider" : "Microsoft",
   "price" : 560.0,
   "technical" : ["snapshot", "distributed"],
   "business" : ["reliability", "recovery", "high_availability",
"performance", "load_balancing"] }
]
```


Σε αυτό το σημείο τα παραπάνω προϊόντα είναι εγγεγραμμένα στο σύστημα. Τώρα, ο κάθε πελάτης που επιθυμεί ένα προϊόν τύπου Filesystem as a Service, απευθύνεται στο σύστημα. Το ερώτημα που θέτει μπορεί να είναι για business χαρακτηριστικά ή για τεχνικά (technical) χαρακτηριστικά ή και για τα δύο. Είναι συχνό και λογικό, οι πελάτες αυτού του συστήματος να γνωρίζουν περισσότερες λεπτομέρειες για χαρακτηριστικά τύπου business και λιγότερες για τύπου technical. Σε κάθε περίπτωση, το σύστημά μας προτείνει τα καταλληλότερα προϊόντα ανάλογα με τα χαρακτηριστικά που ζητάει ο πελάτης, το οποίο και εξηγούμε στη συνέχεια.

Παράδειγμα αποτίμησης προϊόντος

Όπως είδαμε προηγουμένως, το σύστημα εξ αρχής αποτιμάει τα προϊόντα με συγκεκριμένο τρόπο. Η αποτίμηση έγκειται στο όνομα – τη φίρμα – του file system και στα καθαρά τεχνικά του χαρακτηριστικά. Κάθε όνομα και τεχνικό χαρακτηριστικό λαμβάνουν από μια τιμή, τιμή που μπορεί να επηρεάζεται από πολλούς παράγοντες, όντας κυριότερος αυτός της ελεύθερης αγοράς και οικονομίας – ο νόμος της προσφοράς και της ζήτησης. Τα business χαρακτηριστικά δεν λαμβάνουν κάποια τιμή, αλλά δημιουργούνται από την ύπαρξη των technical. Για παράδειγμα:

Το παραπάνω "MooseFS" file system προσφέρεται από τη Microsoft, που δηλώνει ότι περιέχει snapshot και ότι είναι κατανεμημένο (distributed) ως τεχνικά χαρακτηριστικά. Το "MooseFS" αποτιμάται στα 190.0 νομισματικές μονάδες, ενώ τα snapshot και distributed στα 170.0 και 200.0 νομισματικές μονάδες αντίστοιχα. Το προϊόν λαμβάνει συνολική αξία στα 560.0 νομισματικές μονάδες. Το snapshotting προσδίδει σε ένα filesystem τα εξής business χαρακτηριστικά:

- high availability (υψηλή διαθεσιμότητα): είναι κατάλληλο για συστήματα υψηλής διαθεσιμότητας μεγάλων δεδομένων, τα οποία πρέπει αν έχουν online τα δεδομένα τους όλο το 24ωρο. Τέτοια συστήματα προτιμούν να πραγματοποιήσουν backup σε μορφή snapshot, αφήνοντας τους πελάτες τους να συνεχίσουν να γράφουν στα δεδομένα. Αυτό συμβαίνει γιατί στο snapshotting ο χρόνος και οι λειτουργίες εισόδου και εξόδου δεν αυξάνουν με το μέγεθος των δεδομένων, σε αντίθεση με το κλασικό backup όπου είναι απαγορευτικά για συστήματα υψηλής διαθεσιμότητας.
- reliability και recovery (αξιοπιστία και ανάκτηση δεδομένων): μπορεί να χρησιμοποιηθεί ως backup της κατάστασης του file system μια συγκεκριμένη χρονική στιγμή, προσδίδοντας αξιοπιστία (reliability) για την κατάσταση κρίσιμων αρχείων (state) εκείνη τη χρονική στιγμή, και δυνατότητα ανάκτησης αρχείων (recovery) σε περίπτωση βλάβης συστήματος ή άλλου τρόπου αλλοίωσης δεδομένων. Το snapshot είναι πιο γρήγορο από το κανονικό backup.
- storage optimization (αποδοτικότερος χώρος αποθήκευσης): χρησιμοποιεί δείκτες για τα μέρη του αποθηκευτικού χώρου που δεν αλλάζουν δεσμεύοντας νέο χώρο μόνο για τις αλλαγές – αποδοτικότερο από ένα απλό backup.

Από την άλλη το "MooseFS" είναι κατανεμημένο file system και προσδίδει τα εξής business χαρακτηριστικά:

- performance (υπολογιστική απόδοση): αυτόματα λόγω οριζόντιας κλιμάκωσης (horizontal scaling) κάθε κατανεμημένο file system αυξάνει τις αποδόσεις του υπολογιστικά.
- high availability (υψηλή διαθεσιμότητα): για τον ίδιο λόγο της οριζόντιας κλιμάκωσης, αποκτάει υψηλή διαθεσιμότητα δεδομένων.
- load balancing (εξισορρόπηση φορτίου): κάθε κατανεμημένο σύστημα εξ ορισμού κάνει εξισορρόπηση του φόρτου εργασίας.

Βέβαια, το "MooseFS" ως κατανεμημένο file system δεν βοηθάει στην βελτιστοποίηση αποθηκευτικού χώρου:

- storage optimization (αποδοτικότερος χώρος αποθήκευσης): η οριζόντια κλιμάκωση ενός κατανεμημένου file system επεκτείνει τον χώρο αποθήκευσης.

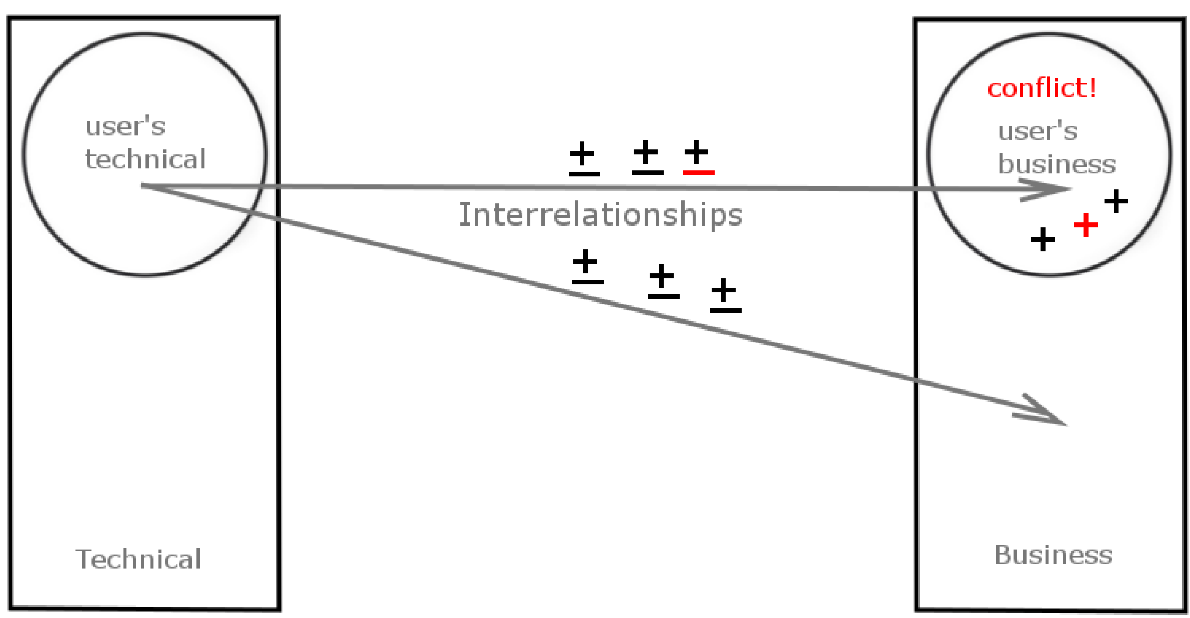
Τελικά, τα τεχνικά του χαρακτηριστικά προσδίδουν για την υπηρεσία μας την λίστα των business χαρακτηριστικών που είδαμε παραπάνω, χωρίς το business attribute "storage optimization":

```
"business" : ["reliability", "recovery", "high_availability", "performance",
"load_balancing"]
```

Αλληλοσυσχετίσεις technical και business

Επανερχόμενοι τώρα στο ερώτημα του χρήστη ο οποίος μπορεί να ζητάει είτε τεχνικά, είτε business, είτε ταυτόχρονα και τα δύο είδη χαρακτηριστικών, διαπιστώνουμε ότι η ανάλυσή του από την υπηρεσία μας γίνεται με παρόμοιο τρόπο. Το σύστημά μας ξεχωρίζει το ερώτημα του χρήστη σε λέξεις-κλειδιά τα οποία αντιστοιχούν είτε σε technical είτε σε business χαρακτηριστικά. Κρατώντας τα technical και τα business ξεχωριστά, επιλύει ποια νέα business προκύπτουν από τα τεχνικά χαρακτηριστικά που έδωσε ο χρήστης. Αυτά ονομάζονται business χαρακτηριστικά λόγω αλληλοσυσχετίσεων και ο χρήστης μπορεί να γνωρίζει ή να μην γνωρίζει ότι τα ζήτησε έμμεσα με την απαίτηση των τεχνικών χαρακτηριστικών. Προκειμένου να ξεκινήσει την αναζήτηση των καλύτερων προϊόντων, η υπηρεσία μας βρίσκει τις διαφορές μεταξύ των business που ο χρήστης ζήτησε άμεσα και αυτών που ζήτησε έμμεσα και κρατάει πρώτα τα έμμεσα – εάν ζήτησε technical ο χρήστης –, και μετά όσα άμεσα δεν συγκρούονται με κάποιο έμμεσο. Έτσι, έχουμε:

- τα τεχνικά (technical) χαρακτηριστικά,
- τα business που ζήτησε ο χρήστης (άμεσα business),
- τα business που προκύπτουν από τα τεχνικά (έμμεσα business) και
- την τομή των δύο τελευταίων (συγκρουόμενα).



Εικόνα 30: Επίλυση αλληλοσυσχετίσεων με παράδειγμα συγκρουόμενων business.

Σε αυτό το σημείο το σύστημα ξέρει τί ζητάει ο χρήστης και θα του το επιστρέψει με συγκεκριμένη προτεραιότητα και σειρά, δίνοντας βάση, όπως θα δούμε, στα τεχνικά χαρακτηριστικά, όπως έχει δηλαδή δομηθεί και η υπηρεσία μας. Κύριο μέλημα είναι να επιστρέφονται πρώτα όσα προϊόντα πληρούν καλύτερα τα κριτήρια του χρήστη (technical και business), και στη συνέχεια να προτείνονται στο χρήστη προϊόντα που είναι σχετικά με αυτά που ζητάει (business μόνο, γιατί τα technical μεταφράζονται όπως είδαμε σε business):

I. Πρώτα επιστρέφει τα προϊόντα εκείνα που εμπεριέχουν ταυτόχρονα όλα τα τεχνικά και business χαρακτηριστικά που ζητάει ο χρήστης. Αυτή η απάντηση είναι το ιδανικό προϊόν ή προϊόντα και συμβαίνει συνήθως όταν απουσιάζουν ειδικές περιπτώσεις αλληλοσυσχετίσεων. Για παράδειγμα, ο χρήστης ζήτησε ένα technical και ένα business, χωρίς το αυτό technical να επηρεάζει αυτό το business, και βρέθηκαν ένα ή περισσότερα προϊόντα που ικανοποιούν το αίτημά του.

II. Σε δεύτερο χρόνο, επιστρέφει τα προϊόντα που εμπεριέχουν όλα τα τεχνικά (technical) χαρακτηριστικά που ζήτησε ο χρήστης, ανεξαρτήτως business. Τα τεχνικά χαρακτηριστικά είναι τα σημαντικότερα στο σύστημά μας: εφόσον, όντας πιο ειδική και συγκεκριμένη τεχνολογία, καλύπτουν καλύτερα τις ανάγκες του χρήστη, καθορίζουν εξ ορισμού την

ύπαρξη των business και έχουν ακρίβεια τιμής. Εάν ζήτησε business που συγκαταλέγονται στα business αλληλοσυσχετίσεων τότε θα υπάρχουν αυτόματα. Αν ζήτησε business που δεν συγκαταλέγονται τότε αυτά παίρνουν κατώτερη προτεραιότητα επειδή σύμφωνα με την υπηρεσία μας, τα τεχνικά χαρακτηριστικά είναι σημαντικότερα. Επομένως, αυτός ο τρόπος επιστρέφει τα καταλληλότερα προϊόντα, μετά τον (I).

III. Στη συνέχεια, το σύστημα επιστρέφει τα προϊόντα που εμπεριέχουν εκείνα τα business χαρακτηριστικά που ζήτησε ο χρήστης αφαιρώντας από αυτά όσα ανήκουν στα συγκρουόμενα, εξασφαλίζοντας ταυτόχρονα ότι δεν θα ξανά-διαλεχτεί κανένα από το (II).

IV. Έπειτα, το σύστημα επιστρέφει τα προϊόντα που εμπεριέχουν εκείνα τα business χαρακτηριστικά που προκύπτουν από αλληλοσυσχετίσεις αφαιρώντας από αυτά όσα ανήκουν στα συγκρουόμενα, εξασφαλίζοντας ταυτόχρονα ότι δεν θα ξανά-διαλεχτεί κανένα από το (III).

Η λίστα των προτεινόμενων από το σύστημα προϊόντων μπορεί να συνεχίζει να επεκτείνεται με ακόμη χαμηλότερης προτεραιότητας προϊόντα. Πιο αναλυτικά, θα δούμε πώς μπορεί να γίνει αυτό προσεχώς, στο Κεφάλαιο 7.

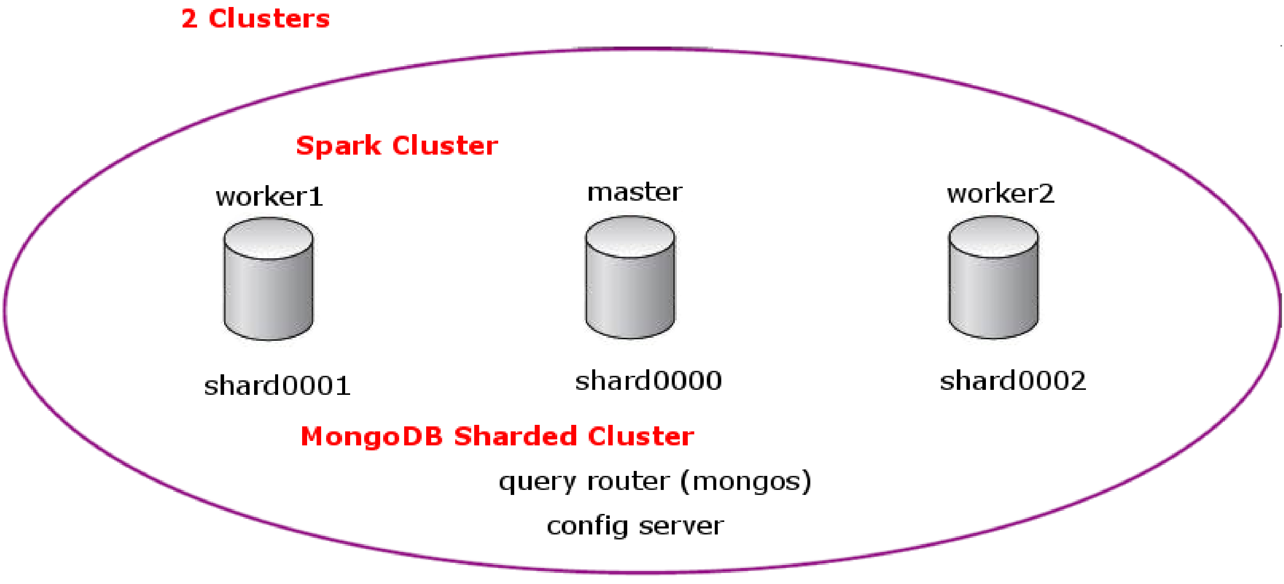
Apache Spark

Επανερχόμενοι τώρα στα κομμάτια της αρχιτεκτονικής αυτά καθαυτά, πρέπει να τονίσουμε ότι όλοι οι παραπάνω υπολογισμοί γίνονται με τη βοήθεια του Apache Spark το οποίο στην περίπτωσή μας αναπτύσσεται με Standalone Scheduler cluster manager. Περισσότερα για διαχειριστές συμπλεγμάτων βλέπουμε στην ομώνυμη παράγραφο του Spark στο Κεφάλαιο 2. Όπως είδαμε, το Apache Spark είναι μια κατανεμημένη υπολογιστική μηχανή κατάλληλη για ανάλυση και παράλληλη επεξεργασία μεγάλων δεδομένων (big data). Το Spark αποτελεί ένα από τα πιο γρήγορα σύγχρονα υπολογιστικά συστήματα συμπλεγμάτων (cluster), καθώς, επεκτείνοντας το μοντέλο MapReduce του Hadoop, μπορεί να λειτουργεί έως πάνω από 100 φορές πιο γρήγορα. Το Spark έχει τη δυνατότητα να μεταφέρει τους υπολογισμούς στη μνήμη (in-memory processing) εκτελώντας την επεξεργασία και ανάλυση σε ταχύτητα RAM. Ακόμη, χρησιμοποιεί μια δική του αφαιρετική δομή δεδομένων, γνωστή ως RDD (Resilient Distributed Datasets), ο χειρισμός της οποίας γίνεται εξαιρετικά αποδοτικά, με βάση την έννοια της οκνηρής αποτίμησης (lazy evaluation). Το Spark ορίζει, όπως είδαμε και στο Κεφάλαιο 2, δύο είδη ενεργειών πάνω σε ένα RDD: τους μετασχηματισμούς (transformations) και τις δράσεις (actions). Lazy evaluation χρησιμοποιεί για κάθε μετασχηματισμό, με την έννοια ότι δεν εκτελεί αμέσως την πράξη του, αλλά την αποθηκεύει για να την εκτελέσει όταν εμφανιστεί η επόμενη δράση (action). Αυτό σημαίνει ότι ένας ή περισσότεροι μετασχηματισμοί μπορούν να αποθηκεύονται στη σειρά για ένα RDD, αναμένοντας

την επόμενη δράση. Το πλεονέκτημα αυτής της ιδιότητας σε συνδυασμό με τις υψηλές ταχύτητες επεξεργασίας αποτελούν τους βασικούς λόγους χρήσης του Spark στην παρούσα εργασία.

Σε δεύτερο χρόνο, λόγοι χρήσης θεωρούνται η ευκολία χρήσης του Spark με τη MongoDB. Όπως θα δούμε και παρακάτω στο Κεφάλαιο 4, προσφέρεται απλή και εύχρηστη διεπαφή για το χειρισμό των δεδομένων στη βάση MongoDB μέσω του Spark, ενώ το Spark επωφελείται από τις ικανότητες του connector MongoDB και Spark, προσθέτοντας στα εργαλεία του "lazy" εντολές που χειρίζονται δεδομένα της MongoDB.

Αρχιτεκτονική components



Εικόνα 31: Αρχιτεκτονική components. Τα "shard000X" αποτελούν τον shard server του κάθε μηχανήματος. Τα εικονικά μηχανήματα που χρησιμοποιήθηκαν έχουν <1.5 GB RAM, 1 CPU, και IPs από αριστερά προς τα δεξιά: 147.102.19.152, 147.102.19.151, 147.102.19.199.

4

Εισαγωγή

Μέχρι στιγμής είδαμε την αρχιτεκτονική του συστήματος στο Κεφάλαιο 3 και αναλύσαμε τα κομμάτια που την απαρτίζουν. Στη συνέχεια, θα δούμε δύο διαφορετικές μεθοδολογίες προσέγγισης του προβλήματος και θα περιγράψουμε τις παραμέτρους αυτών και τον κώδικά τους. Το παρόν Κεφάλαιο εξηγεί και αναλύει τα σημαντικότερα σημεία του κώδικα της εργασίας.

Όπως είδαμε, η παρούσα διπλωματική εργασία παρουσιάζει μια υπηρεσία που προσφέρει στους πελάτες της διαφορετικά προϊόντα τύπου Filesystem as a Service ανάλογα με τις επιθυμίες τους. Υπάρχουν οι πάροχοι των προϊόντων, η υπηρεσία μας, η οποία τα οργανώνει και τα αποτιμάει και οι πελάτες, οι αποδέκτες των προϊόντων. Για το μοντέλο ανάπτυξης της υπηρεσίας, μελετήθηκαν δύο πιθανοί τρόποι, ο ένας βασίζεται στη μέθοδο της Ομοιότητας Συνημιτόνου και απορρίφθηκε σε πρώιμο στάδιο, για λόγους που θα δούμε στη συνέχεια, ενώ ο δεύτερος τρόπος δένει καλύτερα με τη λογική και φιλοσοφία των MongoDB και Spark, και είναι αυτός που χρησιμοποιήθηκε για την υλοποίηση του συστήματος.

Πρώτη Μεθοδολογία

Ξεκινώντας με την πρώτη χρονικά μεθοδολογική προσέγγιση, δίνουμε τον ορισμό[25][26] και τον λόγο που μας έκανε να μελετήσουμε την Ομοιότητα Συνημιτόνου ή Cosine Similarity. Ως ομοιότητα ορίζουμε το πόσο μοιάζουν δύο αντικείμενα όταν τα συγκρίνουμε. Πιο συγκεκριμένα, η Ομοιότητα Συνημιτόνου (Cosine Similarity) αποτελεί ένα μέτρο ομοιότητας μεταξύ δύο μη-μηδενικών διανυσμάτων που βασίζεται και μετράει το συνημίτονο της μεταξύ τους γωνίας. Για δύο διανύσματα **A** και **B**, με $||\mathbf{A}||_2$ και $||\mathbf{B}||_2$ τα μέτρα τους (Ευκλείδεια απόσταση), A_i, B_i ($1 \leq i \leq n$) οι συνιστώσες τους και θ η μεταξύ τους γωνία, η Ομοιότητα Συνημιτόνου υπολογίζεται:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Εικόνα 32: Τύπος υπολογισμού Cosine Similarity

Η Cosine Similarity για δύο διανύσματα με ίδιο προσανατολισμό, δηλαδή γωνία 0°, είναι 1, με κάθετο προσανατολισμό και γωνία 90°, είναι 0, ενώ με αντίθετο προσανατολισμό και γωνία 180°, είναι -1. Από τον παραπάνω ορισμό συμπεραίνουμε ότι τα δύο διανύσματα **A** και **B** μοιάζουν κάτα μία τιμή *cos(θ)*, που παίρνει τιμές από -1 έως 1. Γενικά, η cosine similarity χρησιμοποιείται στον θετικό χώρο όπου οι τιμές της βρίσκονται στο διάστημα [0,1], το οποίο ακριβώς συμβαίνει και στην περίπτωσή μας. Αυτό σημαίνει ότι δύο διανύσματα μπορεί να "μοιάζουν" από 0 (καθόλου) μέχρι 1 (ταύτιση κατεύθυνσης). Υπολογίζοντας τις ομοιότητες συνημιτόνου (cosine similarities) από ζεύγη διανυσμάτων, μπορούμε να γνωρίζουμε με έναν αριθμό (την τιμή similarity) το πόσο μοιάζουν ή όχι αυτά τα διανύσματα, και επιπρόσθετα χρησιμοποιώντας ιδιότητες όπως η προσεταιριστική, να βρίσκουμε ποια άλλα διανύσμα είναι ως ένα βαθμό όμοια με κάποιο συγκεκριμένο. Συνεχίζοντας, θα δούμε πώς αυτό συνδέεται με το πρόβλημά μας και την πρώτη προσέγγιση επίλυσής του.

Η κεντρική ιδέα είναι ότι κάθε file system μπορεί να αντιστοιχηθεί σε ένα διάνυσμα με συνιστώσες τα χαρακτηριστικά[24]. Κάθε συνιστώσα μπορεί να παίρνει τις τιμές 1 ή 0 ανάλογα με το αν το συγκεκριμένο χαρακτηριστικό υπάρχει ή όχι. Μέσω του παρακάτω παραδείγματος θα διακρίνουμε καλύτερα την χρήση Cosine Similarity.

Αν υποθέσουμε ότι τα χαρακτηριστικά που μπορεί να έχει ένα προϊόν του συστήματος είναι μόνο τα "journaling", "snapshot" και "replication", τότε έστω τα τρία παρακάτω προϊόντα της IBM:

- Product 1: HAMMER, FilesystemAsAService, price:280.0 με snapshot
- Product 2: BTRFS, FilesystemAsAService, price:520.0 με journaling, snapshot, replication
- Product 3: GlusterFS, FilesystemAsAService, price:615.0 με snapshot, replication

Για τον υπολογισμό των ομοιοτήτων συνημιτόνου (cosine similarities) μεταξύ των προϊόντων, περιλαμβάνουμε τα χαρακτηριστικά στο κάθε διάνυσμα **A** με συνιστώσες (<journaling>, <snapshot>, <replication>), και με βάση την παραπάνω διαδικασία, έχουμε:

	A			$\|\|\mathbf{A}\|\|_2$	Product 1	Product 2	Product 3
Product 1	0	1	0	1	1		
Product 2	1	1	1	1.73205081	0.577350269	1	
Product 3	0	1	1	1.41421356	0.707106781	0.8164965809	1

Πίνακας 2: Cosine Similarity προϊόντων 1

Με αυτόν τον πίνακα που περιγράφει τις cosine similarities μεταξύ των τριών προϊόντων, μπορούμε ευκολότερα να διακρίνουμε ποια προϊόντα είναι ομοιότερα με ποια άλλα και κατά πόσο πιο όμοια – τα άδεια κελιά έχουν την ίδια τιμή των αντίστοιχων συμμετρικών γεμάτων. Αυτό μας εξυπηρετεί στην περίπτωση του προβλήματός μας, επειδή ο πελάτης της υπηρεσίας μπορεί να ζητάει ένα προϊόν με τον ίδιο τρόπο που ήδη αναφέραμε στο Κεφάλαιο 3, δηλαδή με λέξεις-κλειδιά, στη συνέχεια, το σύστημα να υπολογίζει την cosine similarity ανάμεσα στα διαθέσιμα και στο προϊόν που ζητείται, και στο τέλος να ταξινομεί και να επιστρέφει τις ομοιότητες συνημιτόνου σε φθίνουσα σειρά, προτείνοντας με αυτόν τον τρόπο στον πελάτη τα καταλληλότερα προϊόντα σχετικά με το αίτημά του. Στο συγκεκριμένο παράδειγμα, εάν ο πελάτης ζητήσει ένα προϊόν με journaling και snapshot, τότε το σύστημα δημιουργεί ένα Product 4 με διάνυσμα (1,1,0), υπολογίζει τις cosine similarities με κάθε άλλο διαθέσιμο προϊόν, τις ταξινομεί και επιστρέφει τα προϊόντα με πρώτο εκείνο με τη μεγαλύτερη ομοιότητα συνημιτόνου με το Product 4:

	A			$\|A\|_2$	Product 1	Product 2	Product 3
Product 4	1	1	0	1.41421356	0.7071067812	0.8164965809	0.5

Πίνακας 3: Cosine Similarity προϊόντων 2

Η σειρά που η υπηρεσία προτείνει τα προϊόντα στον πελάτη είναι βέλτιστη, θεωρεί ότι το ιδανικό προϊόν είναι αυτό που ζητάει ο πελάτης, ανεξάρτητα αν θα βρει ταυτόσημο ή όχι (εδώ, δηλαδή, το Product 4), και επιστρέφει τα προϊόντα που είναι ομοιότερα σε αυτό. Προϊόντα με επιπλέον χαρακτηριστικά, τα οποία δεν ζητήθηκαν (για παράδειγμα, εδώ, το replication), θεωρούνται λιγότερο κατάλληλα – αυτό αποδεικνύεται πολύ απλά με την έννοια της αποτίμησης των προϊόντων από την ίδια την υπηρεσία, που αναλύθηκε στο Κεφάλαιο 3 και υποστηρίζει ότι αυτά τα προϊόντα είναι ακριβότερα σε τιμή. Στο παρόν παράδειγμα, ζητείται από τον πελάτη το προϊόν Product 4 με διάνυσμα (1,1,0) ανάμεσα στα διαθέσιμα προϊόντα Product 1 με (0,1,0), Product 2 με (1,1,1), Product 3 με (0,1,1). Όπως βλέπουμε, υπολογίζοντας τις ομοιότητες συνημιτόνου, τα προϊόντα προτείνονται με τη σειρά Product 2, Product 1 και Product 3 ως καταλληλότερα.

Όλα τα παραπάνω υλοποιήθηκαν σε κώδικα ο οποίος μπορεί να βρεθεί στο τέλος του Παραρτήματος 3. Οι μέθοδοι προσέγγισης ενός τέτοιου προβλήματος που χρησιμοποιήθηκαν παραπάνω, σχετίζονται με την Θεωρία των Συστημάτων Προτάσεων[24] (Recommendation Systems) και πιο συγκεκριμένα με το Content-based filtering, που εστιάζει σε συστήματα όπου δημιουργούνται προφίλ πραγμάτων (προϊόντων) και συγκρίνονται μεταξύ τους. Συνεχίζουμε με μια μικρή αναφορά στα κομμάτια του κώδικα και, μετά από αυτό, κλείνουμε την ανάλυση της πρώτης μεθοδολογίας με τους λόγους της πρώιμης απόρριψής της.

Το Scala αντικείμενο ProductRecommender, που περιέχει τη main(), παίρνει την είσοδο του πελάτη (journaling, snapshot), δημιουργεί εσωτερικά το Product 4, δηλαδή (1,1,0), και με τη βοήθεια του Apache Spark:

- Υπολογίζει με τη βοήθεια του ComputeCosSim Scala Object τις cosine similarities του Product 4 με τα άλλα προϊόντα της βάσης και δημιουργεί ζεύγη τύπου **(**similarity P με Product 4**,** P**)**, όπου P προϊόν της βάσης. Εδώ χρησιμοποιεί τη map() που είναι μετασχηματισμός, συνεπώς επωφελούμαστε σε χρόνο από lazy evaluation.
    - Το ComputeCosSim αντικείμενο διαθέτει τις συναρτήσεις computeNorm, που υπολογίζει το μέτρο του διανύσματος (νόρμα) για ένα προϊόν – πρέπει να αναφέρουμε ότι στον πραγματικό κώδικα το διάνυσμα διαθέτει και ένα τέταρτο πεδίο για την τιμή του προϊόντος (price), κάτι που βέβαια δεν αλλάζει την ουσία του παραπάνω παραδείγματος ούτε της Cosine Similarity ανάλυσης – και computeCosSim, που υπολογίζει την ομοιότητα συνημιτόνου μεταξύ του Product 4 και ενός προϊόντος του συστήματος.
- Ταξινομεί τα ζεύγη που δημιούργησε ως προς τις ομοιότητες συνημιτόνου, επίσης lazy η συνάρτηση sortBy() (transformation)
- Τα τυπώνει, με τη δράση (action) foreach(), εμφανίζοντας τα προϊόντα που ταιριάζουν καλύτερα σε αυτό που ζήτησε ο πελάτης

Το Scala αντικείμενο ProductRecommender κάθε φόρα εισάγει και εξάγει από τη βάση τα προϊόντα, πράγμα που οφείλεται στο ότι ο κώδικας ήταν σε testing στάδιο ακόμη όταν

δημιουργήθηκε και περιέχει τέτοιες λεπτομέρειες, όπως και η τιμή (price) στην συνάρτηση computeNorm την οποία είδαμε παραπάνω ή οι τοπικές χρήσεις των MongoDB και Spark. Κάτι τέτοιο δεν μας ενοχλεί βέβαια γιατί δεν αλλοιώνει την ουσία του παραπάνω παραδείγματος ούτε της Cosine Similarity ανάλυσης.

Εφόσον εξηγήσαμε τον τρόπο της πρώτης προσέγγισης και τον λόγο μελέτης της, συνεχίζουμε και κλείνουμε με τα βασικά μειονεκτήματα της μεθοδολογίας αυτής, που μας οδήγησαν σε αλλαγή πορείας.

Ο κύριος λόγος εντοπίζεται στην αρχική μας υπόθεση των τριών χαρακτηριστικών. Ένα τέτοιο σύστημα έχει απαιτήσεις για πολλά περισσότερα χαρακτηριστικά από την απλή τριάδα (<journaling>, <snapshot>, <replication>). Βέβαια, η υπόθεση που έγινε παραπάνω έχει βάση και εφαρμόζεται για συγκεκριμένα χαρακτηριστικά, τα οποία πρέπει να είναι δηλωμένα εξ αρχής στον κώδικα και η τροποποιήση των οποίων απαιτεί αλλαγές σε αυτόν. Αυτά τα χαρακτηριστικά μπορούν να αποτελούν είτε τριάδα είτε ν-άδα. Όμως, σε κάθε περίπτωση που θα επεκτείναμε το παραπάνω σύστημα ώστε να δουλεύει για όσα χαρακτηριστικά χρειάζεται – έστω ν, θα ήταν αναγκαίο κάθε προϊόν που έχει, για παράδειγμα, 2 χαρακτηριστικά, να συνοδεύεται στον κώδικα από ν-2 μηδενικά (1,1,0,…,0), καταλαμβάνοντας περιττά, σε μεγάλη κλίμακα, θέσεις μεταβλητών και μνήμη. Η αντιμετώπιση που διατυπώνουμε προσεχώς, χρησιμοποιεί σαν συμβολοσειρές όσα χαρακτηριστικά περιέχει ένα προϊόν, δηλώνοντας την ύπαρξή τους με αυτόν τον τρόπο, χωρίς επιπλέον περιττές μεταβλητές και υπολογισμούς (όπως για παράδειγμα οι νόρμες). Με αυτόν τον νέο τρόπο η παραπάνω ν-άδα, του προϊόντος με τα 2 από τα ν χαρακτηριστικά, γίνεται (journaling, snapshot), δηλαδή εμπεριέχει μόνο τα χαρακτηριστικά που υπάρχουν στο προϊόν. Η νέα μέθοδος – η δεύτερη προσέγγιση – θα αναπτυχθεί με περισσότερες λεπτομέρειες στη συνέχεια.

Για τους παραπάνω λόγους δεν συνεχίστηκε η μελέτη επίλυσης με τις ομοιότητες συνημιτόνου (cosine similarities) – πρώτη μεθοδολογία – γεγονός που μας έκανε να προσεγγίσουμε διαφορετικά το πρόβλημα. Συνεχίζουμε τώρα, χωρίς επιπλέον αναβολές, στην δεύτερη προσέγγιση.

Δεύτερη Μεθοδολογία

Generator

Όπως είδαμε μέχρι στιγμής, υπάρχει ανάγκη για μια προσέγγιση πιο συγκεκριμένη και πιο ελαστική από την προηγούμενη. Χρειάζεται να είναι πιο συγκεκριμένη προκειμένου να μην γίνονται περιττοί υπολογισμοί και περιττή χρήση μεταβλητών, και πιο ελαστική ώστε το είδος και ο αριθμός των χαρακτηριστικών να μπορεί να είναι δυναμικά, χωρίς αλλαγές στον ίδιο τον κώδικα. Μελετήσαμε το πρόβλημα και καταλήξαμε σε ορισμένες κινήσεις-κλειδιά, τις οποίες έχουμε αναφέρει ήδη στα Κεφάλαια 2 και 3:

- κατηγοριοποιήσαμε τα χαρακτηριστικά σε technical και business
- ορίσαμε κάθε προϊόν να εμπεριέχει, σαν έγγραφο της βάσης δεδομένων MongoDB, μόνο τα χαρακτηριστικά που υποστηρίζει
- ορίσαμε τα business χαρακτηριστικά ενός προϊόντος έτσι ώστε να επηρεάζονται θετικά ή αρνητικά από ένα ή περισσότερα τεχνικά του χαρακτηριστικά. Για παράδειγμα, στην περίπτωση ενός προϊόντος όπου ένα technical δρα θετικά για το ίδιο business

χαρακτηριστικό που άλλα technical δρουν αρνητικά, προσθαφαιρούνται οι τιμές των θετικών και αρνητικών δράσεων και το business χαρακτηριστικό μένει στην εγγραφή του προϊόντος μόνο αν το αποτέλεσμα είναι καθαρά θετικό – σε διαφορετική περίπτωση (αρνητικό ή μηδέν) διαγράφεται.

Σε αυτό το σημείο υπενθυμίζουμε ότι έχουμε ορίσει την τιμή του κάθε προϊόντος με τέτοιο τρόπο ώστε να επηρεάζεται από το όνομα του file system και τα τεχνικά (technical) χαρακτηριστικά. Το όνομα ενός file system – η φίρμα – αποτιμάται από το σύστημά μας με μια τιμή όπως και κάθε τεχνικό χαρακτηριστικό. Η συνολική τιμή ενός προϊόντος προκύπτει από το άθροισμα της τιμής του ονόματος και των τιμών των τεχνικών χαρακτηριστικών που αυτό περιέχει. Η αποτίμηση αυτή, όπως αναφέραμε στα προηγούμενα κεφάλαια, είναι συνάρτηση της προσφοράς και ζήτησης ενός προϊόντος – νόμος της ελεύθερης αγοράς και οικονομίας. Στη συνέχεια, μπορούμε να δούμε τις τιμές που αποδίδουμε στα παραπάνω με τον γεννήτορα προϊόντων (fsGenerator.sh στο Παράρτημα 2) που κατασκευάσαμε:

FS Name	Price
HAMMER	110.0
BTRFS	80.0
ReFS	130.0
Ext4	70.0
ZFS	150.0
APFS	230.0
eCryptFS	70.0
RozoFS	75.0
XFS	60.0
MooseFS	190.0
GlusterFS	195.0
HDFS	210.0

Attribute Name	Price
journaling	100.0
encryption	200.0
distributed	200.0
compression	160.0
erasure_coding	350.0
replication	250.0
cow	50.0
b-tree	60.0
b+tree	70.0
defragmentation	150.0
deduplication	190.0
snapshot	170.0
checksum	110.0
RAID	200.0
mirroring	300.0
scrubbing	290.0
stripping	310.0

Πίνακας 4 και Πίνακας 5: Πίνακες ονομάτων file system – φίρμας – και τεχνικών χαρακτηριστικών – technical attributes – με τις αντίστοιχες τιμές που τους αποδίδει η πλατφόρμα.

Ο bash script γεννήτορας προϊόντων κρατάει σε 2 ξεχωριστούς πίνακες τις στήλες FS Name και Price του πρώτου πίνακα, και δημιουργώντας τυχαίους αριθμούς επιλέγει κάθε φορά ένα όνομα από την πρώτη στήλη, γνωρίζοντας αυτόματα και την τιμή του, εφόσον έχει τον ίδιο δείκτη στον αντίστοιχο πίνακα Price. Με παρόμοιο τρόπο βρίσκει την αντίστοιχη τιμή ενός attribute όταν την ζητήσει κατά τη δημιουργία ενός προϊόντος. Ο γεννήτορας έχει ακόμη δυσδιάστατους πίνακες της μορφής:

```
#ReFS
FilesystemAttr[2,0]="deduplication"
FilesystemAttr[2,1]="scrubbing"
FilesystemAttr[2,2]="checksum"
```

```
FilesystemAttr[2,3]="b+tree"
```

Η θέση 2 (πρώτος δείκτης) αντιστοιχεί στον FS Name πίνακα, εδώ στο ReFS, και ο υπο-πίνακας `FilesystemAttr[2,j]` σημαίνει ότι το file system στη θέση 2 του FS Name, μπορεί να έχει ως τεχνικά χαρακτηριστικά τουλάχιστον μία από τις τιμές `FilesystemAttr[2,0]`, `FilesystemAttr[2,1]`, `FilesystemAttr[2,2]`, και `FilesystemAttr[2,3]`, και το πολύ και τις τέσσερις τιμές. Με αυτόν τον τρόπο ο γεννήτορας δημιουργεί νέα προϊόντα. Στη συνέχεια, γίνεται μία αναφορά στον συντακτικό αναλυτή μετατροπής YAML σε JSON ως κομμάτι του κώδικα, μιας και επεξήγησή του έχει προηγηθεί στο Κεφάλαιο 3.

Συντακτικός Αναλυτής

Ο συντακτικός αναλυτής που κατασκευάσαμε, είδαμε ότι μετατρέπει την έξοδο του γεννήτορα μορφής YAML σε μορφή JSON. Η λειτουργία του βασίζεται στην απλή ανάγνωση των τιμών του κάθε Filesystem As A Service προϊόντος και αντιστοίχισης αυτών στα κατάλληλα JSON πεδία. Σημαντικό σημείο είναι ότι ο αναλυτής διαβάζει αποκλειστικά τα προϊόντα τύπου Filesystem As A Service, χωρίς περιττές αναγνώσεις, από ένα αρχείο που υπακούει στο πρότυπο TOSCA Simple Profile in YAML version 1.1 και το οποίο ο πάροχος δημιουργεί σε κάθε περίπτωση – με τη λεπτομέρεια ότι η μορφή των προϊόντων μέσα σε αυτό τυπώνονται με τη λογική που είδαμε στην αρχή του Κεφαλαίου 3. Εφόσον, όλα τα JSON που αντιστοιχούν στους παρόχους εισαχθούν στη βάση δεδομένων MongoDB, η υπηρεσία είναι έτοιμη να απαντήσει σε αιτήματα πελατών.

Πελάτης

Όλη η περιγραφή και ανάλυση του κώδικα της εφαρμογής επιχειρείται να πλησιάζει τη χρονολογική σειρά εκτέλεσης όσο είναι δυνατόν. Η εφαρμογή ξεκινάει με τον πελάτη να εισάγει λέξεις-κλειδιά σχετικές με technical ή business χαρακτηριστικά που επιθυμεί να έχει το προϊόν Filesystem As A Service που αναζητάει.

Αυτό συμβαίνει στο Scala Object ProductRecommender που περιέχει τη main(). Ο parser της απάντησής του είναι ένα απλό scala.io.StdIn.readLine().toString() της Scala που επιστρέφει string. Το ProductRecommender Object επεξεργάζεται την απάντηση, ξεχωρίζει τις λέξεις-κλειδιά σε technical και business, απορρίπτοντας τις άκυρες, αυτά μέσα στη μέθοδο userAnswer(). Στη συνέχεια, ξεχωρίζει

- τα τεχνικά που ζήτησε ο πελάτης,
- τα business που ζήτησε ο πελάτης, και
- τα business που ζήτησε έμμεσα ο πελάτης μέσω των τεχνικών (αν υπάρχουν). Στην περίπτωση που υπάρχουν, υπολογίζονται με το Scala Object Interrelationships. Ο τρόπος λειτουργίας αυτού του αντικειμένου κρίνεται απαραίτητος για την κατανόηση της συνέχειας της εφαρμογής, γι' αυτό και αναλύεται αμέσως τώρα.

Interrelationship Scala Object

Σε αυτό το αντικείμενο γίνεται ο υπολογισμός των αλληλοσυσχετίσεων των τεχνικών και business χαρακτηριστικών, η θεωρία και η ουσία που αποτελούν τον πυλώνα της όλης εφαρμογής. Όπως είδαμε και αναλύσαμε στο Κεφάλαιο 3, κάθε business χαρακτηριστικό προκύπτει άμεσα από κάποιο τεχνικό χαρακτηριστικό ενός προϊόντος. Στο αντικείμενο Interrelationship που κατασκευάσαμε γίνεται αυτή η πράξη μέσω της συνάρτησης interr(). Αυτή η συνάρτηση παίρνει ως

είσοδο μία λίστα από τεχνικά (technical) χαρακτηριστικά ενός προϊόντος, και για κάθε ένα από αυτά προσθέτει ή αφαιρεί τα business χαρακτηριστικά που αυτό επηρεάζει θετικά ή αρνητικά αντίστοιχα. Επομένως, η interr() συνάρτηση καταλήγει σε μια λίστα της μορφής:

[(b.a.1, 2), (b.a.2, –1), (b.a.3, 0), (b.a.4, 1)] , όπου b.a. = business attribute, για παράδειγμα:

[(performance, 2), (security, –1), (storage optimization, 0), (reliability, 1)]

Αυτή η λίστα σημαίνει ότι όλα τα τεχνικά χαρακτηριστικά του προϊόντος, ύστερα από προσθαφαιρέσεις, επηρεάζουν:

i θετικά την performance,
ii αρνητικά την security,
iii κανέναν αντίκτυπο στο storage optimization,
iv θετικά την reliability

Το κάθε τεχνικό (technical) χαρακτηριστικό μπορεί να αυξάνει ή να μειώνει κατά κάποια τιμή ένα ή περισσότερα business χαρακτηριστικά. Εφόσον, για κάθε business χαρακτηριστικό προσθαφαιρεθούν οι τιμές που δίνονται από τα τεχνικά (technical) , η συνάρτηση interr() καταλήγει και επιστρέφει μια τέτοιου είδους λίστα που είδαμε παραπάνω, από την οποία το ProductRecommender αντικείμενο θα αφαιρέσει σε πρώτο χρόνο όσα χαρακτηριστικά έχουν τιμή μη θετική.

Συνεχίζοντας την επεξεργασία του αιτήματος του πελάτη, το ProductRecommender αντικείμενο προσθέτει άλλη μία κατηγορία που αποτελείται από εκείνα τα business χαρακτηριστικά που συγκρούονται μεταξύ τους στις δύο τελευταίες κατηγορίες (conflicting). Συγκρούονται σημαίνει ότι, ενώ ζητούνται άμεσα από το χρήστη, για παράδειγμα "security", και αν η παραπάνω λίστα είναι αυτή που επιστρέφει η interr(), τότε υπολογίζεται συνολικό security ίσο με το μηδέν. Αυτό οφείλεται σε κάποιο technical χαρακτηριστικό που ζήτησε ταυτόχρονα με το "security" ο χρήστης, με αρνητική επίδραση στο "security", και το οποίο τεχνικό χαρακτηριστικό η υπηρεσία μας λαμβάνει υπόψιν περισσότερο από τα business, επειδή όπως είδαμε κατ' επανάληψη στο Κεφάλαιο 3, τα technical (τεχνικά) χαρακτηριστικά είναι σημαντικότερα, εφόσον, αποτελούν πιο ειδική και συγκεκριμένη τεχνολογία, καλύπτουν καλύτερα τις ανάγκες του χρήστη και καθορίζουν εξ ορισμού την ύπαρξη των business. Σε αυτό το σημείο έχουμε ξεχωρίσει το αίτημα του πελάτη στα παρακάτω χαρακτηριστικά:

- τα τεχνικά που ζήτησε ο πελάτης (αν υπάρχουν),
- τα business που ζήτησε άμεσα ο πελάτης (αν υπάρχουν), αφαιρώντας τα conflicting (αν υπάρχουν)
- τα business που ζήτησε έμμεσα ο πελάτης μέσω των τεχνικών, αφαιρώντας τα conflicting (αν υπάρχουν)

Το αίτημα ενός πελάτη δεν χωρίζεται πάντα σε όλες τις παραπάνω κατηγορίες. Σε περίπτωση, που ζητήθηκαν μόνο business χαρακτηριστικά, δεν υπάρχουν ούτε τα άλλα δύο είδη ούτε τα conflicting. Ακόμη, σε περίπτωση που στο αίτημα υπάρχουν και technical και business, δεν είναι απαραίτητη η ύπαρξη conflicting. Όλα αυτά συμπεραίνονται εύκολα από τον ορισμό και το παράδειγμα του Interrelationship αντικειμένου που δώσαμε παραπάνω. Η περιγραφή του συστήματος συνεχίζει με την σύνδεση του ProductRecommender Scala Object με το Apache Spark και τη MongoDB σε ένα σημείο στον κώδικα.

Apache Spark και MongoDB

Μέχρι στιγμής αναλύσαμε με ποιον τρόπο το σύστημα επεξεργάζεται το αίτημα ενός πελάτη και τους λόγους για τους οποίους συμβαίνει αυτό. Όπως είδαμε, το αίτημα διαχωρίζεται σε τρεις κατηγορίες χαρακτηριστικών: τα τεχνικά που ζήτησε ο πελάτης, τα business που ζήτησε άμεσα ο πελάτης αφαιρώντας τα conflicting από αυτά, και τα business που ζήτησε έμμεσα ο πελάτης μέσω των τεχνικών χαρακτηριστικών αφαιρώντας τα conflicting από αυτά. Στη συνέχεια, το ProductRecommender αντικείμενο συνδέει, με την παρακάτω εντολή, το Apache Spark με τη βάση MongoDB φορτώνοντας σε ένα RDD (Resilient Distributed Dataset) τις πληροφορίες για τη μορφοποίηση του Spark (configuration), οι οποίες εξηγούνται λεπτομερώς στη συνέχεια και οι οποίες περιλαμβάνουν επιπλέον τον τρόπο σύνδεσης με τη MongoDB, και τα έγγραφα της συγκεκριμένης συλλογής στην οποία συνδέεται. Βέβαια, σε αυτό το σημείο τονίζουμε ότι αυτή η πράξη είναι "lazy", πράγμα που, όπως είδαμε εκτενώς στα Κεφάλαια 2 και 3, δεν περιλαμβάνει ούτε υπολογισμούς, ούτε πραγματική φόρτωση όλης της συλλογής των εγγράφων στο RDD:

`var rdd = MongoSpark.load(sc)`, όπου "sc" το SparkContext.

Με τα παραπάνω, φτάνουμε στην αναγκαία εξήγηση του Scala αντικειμένου ClusterSparkContext το οποίο βοηθάει στην ορθή σύνδεση όλων των components ώστε να συνεχίσει το ProductRecommender αντικείμενο την εκτέλεση του προγράμματος.

ClusterSparkContext

Το αντικείμενο αυτό χειρίζεται το configuration του Spark και οτιδήποτε αυτό περιλαμβάνει. Δημιουργεί ένα SparkSession[27] και το χρησιμοποιεί ως σημείο επικοινωνίας του κώδικα (αντικείμενο ProductRecommender) με το ίδιο το Spark. Μέσα σε αυτό ορίζεται ολόκληρο το configuration:

- Ο master κόμβος του Spark, με τη συνάρτηση `.master("spark://master:7077")`. Με αυτόν τον τρόπο συνδέει το SparkSession με τον master κόμβο, ο οποίος διαχειρίζεται τους workers και όλο το Spark Ecosystem.

- Το όνομα της εφαρμογής που υποβάλλεται στο Spark, συνάρτηση `.appName("Product Recommender System")`

- Ο φάκελος όπου ο master θα βρει το αρχείο JAR της εφαρμογής ώστε να το διανέμει στους workers, συνάρτηση:
  `.config("spark.jars", "/home/kapsoulis/ProductRecommenderSystem.jar")`
  Οι workers του συμπλέγματος τρέχουν αυτοί Spark, όπως ο master, επομένως πρέπει να γνωρίζουν την ίδια την εφαρμογή. Αυτό γίνεται με ευθύνη του master, ο οποίος διανέμει το JAR της εφαρμογής τη στιγμή εκκίνησης του cluster.

- Η σύνδεση των URIs
  εισόδου: `.config("spark.mongodb.input.uri",s"mongodb://${USERNAME}:${PASSWORD}@${SERVER}:${PORT}/${DATABASE}.${COLLECTION}?authSource=admin")`
  και εξόδου: `.config("spark.mongodb.output.uri",s"mongodb://${USERNAME}:${PASSWORD}@${SERVER}:${PORT}/${DATABASE}.${COLLECTION}?authSource=admin")`
  δεδομένων από τη βάση MongoDB. Οι μεταβλητές με τους κεφαλαίους λατινικούς χαρακτήρες έχουν πάρει ήδη τιμή και είναι

- ◦ USERNAME, PASSWORD: τα credentials σύνδεσης στην διαχειριστική βάση admin
  - ◦ SERVER, PORT: η IP και η θύρα του mongos κόμβου (στην εφαρμογή μας έχουμε σηκώσει έναν query router)
  - ◦ DATABASE, COLLECTION: η βάση και η συγκεκριμένη συλλογή της στην οποία συνδέεται το SparkContext.
- Η γνωστοποίηση στο Spark ότι η συλλογή COLLECTION της βάσης MongoDB είναι partitioned με sharding:
  `.config("spark.mongodb.input.partitioner", "MongoShardedPartitioner$")`
  και με ποιο κλειδί έγινε sharded η συλλογή:
  `.config("spark.mongodb.input.partitionerOptions.shardKey", "price")`

Επιπλέον, είναι σημαντικό να αναφέρουμε την τελευταία εντολή του SparkSession, την `.getOrCreate()`, η οποία επιστρέφει ένα υπάρχον SparkSession, αν υπάρχει ήδη, αλλιώς δημιουργεί το νέο. Σε κάθε περίπτωση, εφαρμόζει τις παραπάνω μορφοποιήσεις (configuration), είτε στο υπάρχον είτε στο νέο SparkSession. Αυτό συμβαίνει επειδή σε κάθε περίπτωση πρέπει να υπάρχει μόνο ένα SparkContext ανά JVM, όπως εξηγήσαμε στο Κεφάλαιο 2. Το SparkContext είναι ουσιαστικά πεδίο του SparkSession και η προσπέλασή του γίνεται μέσω του SparkSession:

Ορισμός στο ClusterSparkContext αντικείμενο:
`val spark = SparkSession.builder()`, μετά τον builder ακολουθεί κάθε configuration του παραπάνω ορισμού.

Προσπέλαση στο ProductRecommender:
`val sc = ClusterSparkContext.spark.sparkContext`, προσπέλαση SparkContext.

Σε αυτό το σημείο και εφόσον είδαμε τον τρόπο σύνδεσης όλων των components στον κώδικα, συνεχίζουμε από την εκτέλεση του προγράμματος από το ProductRecommender αντικείμενο, δηλαδή μετά τον "lazy" μετασχηματισμό `load`:

`var rdd = MongoSpark.load(sc)`, όπου "sc" το SparkContext.

Το Scala Object ProductRecommender εφαρμόζει τώρα έναν ακόμη lazy μετασχηματισμό που μορφοποιεί το RDD:

```
val aggregatedRdd = rdd.withPipeline(Seq(Document.parse(..)))
```

Η συνάρτηση `withPipeline` (lazy), εκμεταλλευόμενη τον μηχανισμό Aggregation Pipeline της MongoDB (βλ. Παράρτημα 1), φιλτράρει τα έγγραφα και εκτελεί aggregations στο – τύπου MongoRDD – `rdd`, προτού περαστούν τα έγγραφα στο Spark. Αυτό το φιλτράρισμα αποδέχεται μόνο έγγραφα που είναι σχετικά με μία από τις τρεις κατηγορίες χαρακτηριστικών στις οποίες ξεχωρίσαμε προηγουμένως το αίτημα του πελάτη:

- τα τεχνικά που ζήτησε ο πελάτης,
- τα business που ζήτησε άμεσα ο πελάτης αφαιρώντας τα conflicting, και

- τα business που ζήτησε έμμεσα ο πελάτης μέσω των τεχνικών αφαιρώντας τα conflicting

Αυτό γίνεται ώστε, στην πρώτη δράση (action) που θα ακολουθήσει, τα έγγραφα της βάσης που θα φορτωθούν στην πραγματικότητα στο νέο RDD να είναι λιγότερα σε αριθμό. Χωρίς τη συνάρτηση `withPipeline`, το Spark θα φόρτωνε τα έγγραφα ολόκληρης τη συλλογής.

Στη συνέχεια, το ProductRecommender αντικείμενο εφαρμόζει την παρακάτω δράση στο εναπομείνον RDD:

```
aggregatedRdd.foreach{ doc ⇒  … }
```

Πριν προχωρήσουμε, είναι αναγκαίο να αναφέρουμε τη χρήση του accumulator "`seqDocAccum`". Όταν μια συνάρτηση περνάει σε μια μέθοδο του Spark (όπως map ή reduce) και εκτελείται σε απομακρυνσμένους κόμβους, κάθε κόμβος worker χρησιμοποιεί ένα αντίγραφο των μεταβλητών της συνάρτησης των οποίων τις τιμές δεν μαθαίνει ο driver. Έχοντας ανάγκη από μια μεταβλητή η οποία θα διατηρεί την τιμή της ανάμεσα στους workers, το Spark προσφέρει την μεταβλητή είδους accumulator. Εμείς χρησιμοποιώντας αυτή τη λογική, αναπτύξαμε το δικό μας accumulator σε μορφή λίστας ζευγών ώστε να ικανοποιείται η περίπτωση του προβλήματος που λύνουμε. Περισσότερα για το είδος της μεταβλητής accumulator φαίνονται στο Παράρτημα 1.

Τα έγγραφα που πέρασαν ως "έγκυρα" το προηγούμενο φιλτράρισμα της `withPipeline`, διαχωρίζονται τώρα μέσα στη `foreach` με βάση τα κριτήρια που αναλύσαμε στο τέλος του Κεφαλαίου 3, εν ονόματι I, II, III και IV. Η `foreach` κρατάει στον accumulator "`seqDocAccum`" μια λίστα ζευγών τύπου (document, integer) τέτοια ώστε κάθε έγγραφο (προϊόν) που ικανοποιεί το κριτήριο:

- (I), να βρίσκεται στη λίστα ως (προϊόν, 0),
- (II), να βρίσκεται στη λίστα ως (προϊόν, 1),
- (III), να βρίσκεται στη λίστα ως (προϊόν, 2),
- (IV), να βρίσκεται στη λίστα ως (προϊόν, 3).

Αυτό φαίνεται στον κώδικα στην συνθήκη μέσα στη `foreach`:

```
 if (techBool && busiBool) // ideal product!!
         seqDocAccum.add(Seq((doc,0)))
       else if (techBool) // if this product has ALL the technical the user wants
         seqDocAccum.add(Seq((doc,1)))
       else if (busiBool) // if this product has ALL the business the user wants
         seqDocAccum.add(Seq((doc,2)))
       else if (interrBool)
         seqDocAccum.add(Seq((doc,3)))
```

Βλέπουμε ότι οι ακέραιοι αριθμοί ουσιαστικά αντιπροσωπεύουν την προτεραιότητα των προϊόντων (υπενθυμίζουμε ότι το πρώτο κριτήριο (I) έχει την υψηλότερη προτεραιότητα).

Ο accumulator "`seqDocAccum`" συνεχίζει ταξινομώντας τα ζεύγη με βάση τις προτεραιότητες ώστε τα προϊόντα με την υψηλότερη προτεραιότητα να βρίσκονται πρώτα σε σειρά και ώστε τα προϊόντα με την ίδια προτεραιότητα να εμφανίζονται σε φθίνουσα σειρά ως προς την τιμή (price).

Σε αυτό το σημείο, το ProductRecommender αντικείμενο προτείνει στον πελάτη τα πρώτα 25 καταλληλότερα προϊόντα βάσει του αιτήματός του, εκτυπώνοντας τον accumulator:

```
    seqDocAccum.value.filter(x ⇒ !x._1.isEmpty()).sortBy(s ⇒ (s._2,
s._1.getDouble("price")) ).take(25).foreach(x⇒
        println(x._1.getString("name")+"\t"+
                x._1.getDouble("price")+"\t"+
                x._1.getString("provider")+"\t"+
                x._1.getString("service_type")+"\t"+
                x._1.get("technical")+
                x._1.get("business"))
    )
```

.

Στο Κεφάλαιο αυτό αναλύσαμε εκτενώς τις δύο μεθοδολογίες προσέγγισης του προβλήματος. Η πρώτη όπως είδαμε απορρίφθηκε νωρίς στην υλοποίησή της για συγκεκριμένους λόγους, ενώ η δεύτερη είναι αυτή που λύνει το πρόβλημα και υλοποιεί την εφαρμογή. Στο κομμάτι της δεύτερης μεθοδολογίας αναλύθηκαν τα βασικότερα σημεία του κώδικα και λεπτομέρειες πίσω από την υλοποίησή του. Στο επόμενο κεφάλαιο θα δούμε στατιστικά αποτελέσματα μετρήσεων της απόδοσης της εφαρμογής.

5

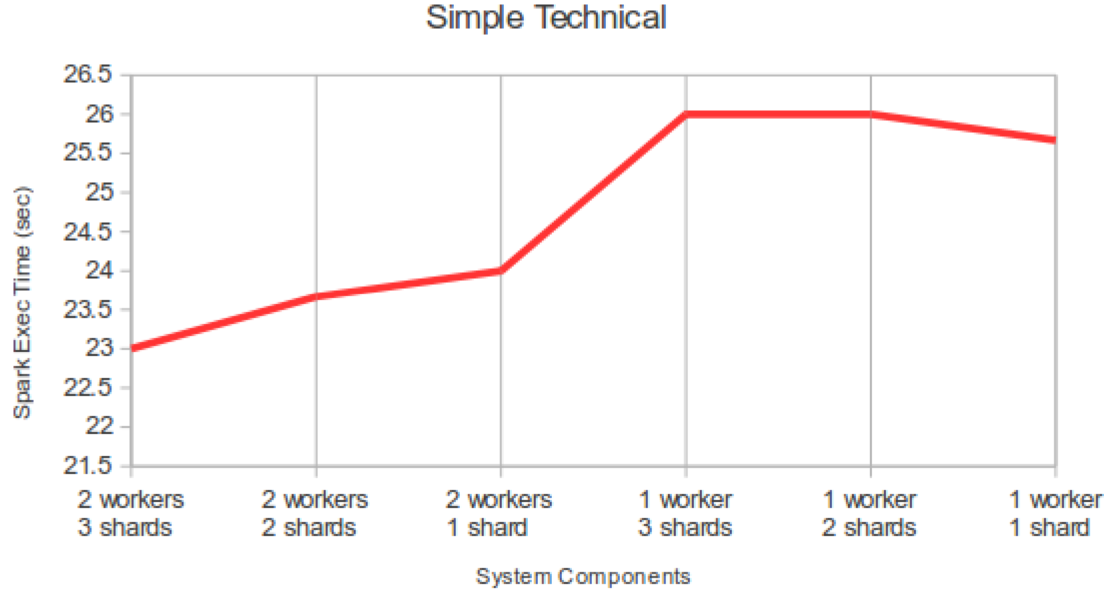
Εισαγωγή

Στο παρόν κεφάλαιο περιγράφουμε και αναλύουμε πληθώρα μετρήσεων τις οποίες πήραμε εκτελώντας την εφαρμογή μας. Γίνεται παρουσίαση στατιστικών αποτελεσμάτων με τη βοήθεια διαγραμμάτων και ορισμένων τρόπων λειτουργίας του συστήματος με τη βοήθεια εικόνων. Υποβάλλουμε διαφορετικά αιτήματα ενός πελάτη σε διαφορετικές μορφοποιήσεις των components του συστήματος και παίρνουμε μετρήσεις της χρονικής επεξεργασίας των αιτημάτων αυτών. Ακολουθούν αναλυτικά τα αποτελέσματα και οι αναλύσεις επί των μετρήσεων.

Υποβάλλοντας διαφορετικά αιτήματα πελάτη σε διαφορετικές μορφοποιήσεις των components του συστήματος Apache Spark και MongoDB, πήραμε μετρήσεις επί του χρόνου επεξεργασίας των αιτημάτων με τη βοήθεια του Spark Standalone Scheduler, δηλαδή του cluster manager που χρησιμοποιούμε. Περισσότερα για διαχειριστές συμπλεγμάτων (cluster managers) του Spark αναφέρουμε στην ομώνυμη παράγραφο του Κεφαλαίου 2. Ο Standalone Scheduler μετράει τον χρόνο εκτέλεσης της κάθε υποβληθείσας εφαρμογής και τον εμφανίζει στο Spark Web UI.

Παρακάτω παραθέτουμε τα αποτελέσματα του χρόνου απόκρισης του Spark για εννέα διαφορετικά ερωτήματα σε 100.000 προϊόντα.

Simple Technical

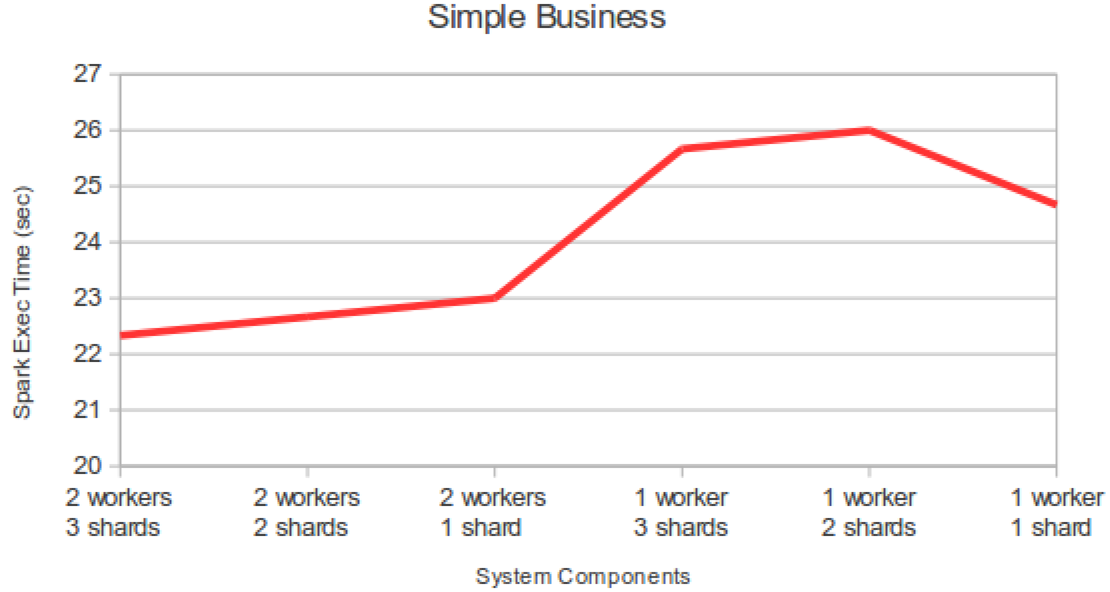


Πίνακας 6: Simple Technical Query

Αυτό το αίτημα πελάτη ονομάζεται "Simple Technical" και αποτελείται από τις λέξεις-κλειδιά "cow". Ο πελάτης ζητάει από το σύστημά μας να του επιστρέψει τα καλύτερα προϊόντα που περιέχουν copy-on-write (cow). Το σύστημα βρίσκει αυτά τα προϊόντα, επιλύει τις αλληλοσυσχετίσεις – δηλαδή, στην προκειμένη περίπτωση, εντοπίζει επιπλέον όσα προϊόντα δεν ανήκουν στην προηγούμενη κατηγορία και περιέχουν εκείνα τα business χαρακτηριστικά που δίνει το τεχνικό χαρακτηριστικό "cow" – και επιστρέφει τη λίστα των προτεινόμενων προϊόντων στον χρήστη. Όπως βλέπουμε στο διάγραμμα, το ίδιο αίτημα υποβλήθηκε στην εφαρμογή για διαφορετικές μορφοποιήσεις των components της πλατφόρμας. Ρυθμίζοντας τους workers του Spark σε 1 ή 2 και ταυτόχρονα τα shards της MongoDB σε 1 ή 2 ή 3 καταλήξαμε σε 6 περιπτώσεις, όπως φαίνονται στο παραπάνω διάγραμμα. Παρατηρούμε ότι όσο πιο κατανεμημένα δουλεύουν το Spark και η MongoDB τόσο πιο γρήγορη χρονική απόκριση έχει η εφαρμογή μας. Όταν το Spark τρέχει με 2 workers ο driver αναθέτει tasks στους executors του κάθε worker, και ο καθένας από αυτούς τρέχει το task ως μία διεργασία (process). Στην περίπτωση που το Spark τρέχει με έναν worker κόμβο, όλα τα tasks θα ανατεθούν μόνο στους executors αυτού, με αποτέλεσμα να υποχρεώνεται να φέρει εις πέρας διπλάσια στον αριθμό tasks, φυσικά με κόστος σε χρόνο, όπως βλέπουμε. Βέβαια, η κατανεμημένη εκτέλεση σε Spark Cluster περιβάλλον δεν αποτελεί την μοναδική αιτία της καλύτερης χρονικής απόκρισης της εφαρμογής. Σημαντικότατος παράγοντας είναι και η συμβολή του sharding της MongoDB στο σύστημα, ένα ακόμη Cluster περιβάλλον της πλατφόρμας. Την ώρα που το Spark χρησιμοποιεί μέγιστα 2 εικονικά μηχανήματα για το cluster του, η MongoDB

χρησιμοποιεί 3, αναθέτοντας από ένα shard στο κάθε μηχάνημα. Επομένως, το κάθε ερώτημα που τίθεται στον query router (mongos) – όταν η MongoDB λειτουργεί με Sharded Cluster τριών (3) shards – επωφελείται του sharding σε χρόνο επιστροφής της απάντησης. Ο query router είναι κατάλληλος για την δρομολόγηση ερωτημάτων και λειτουργιών (write, update) στα σωστά shards, επιτυγχάνοντας καλύτερη χρονική απόκριση στα αιτήματα του συστήματος. Όπως είδαμε, αυτό το πετυχαίνει φέρνοντας στην cache τα metadata των δεδομένων από τον configuration server. Περισσότερα για το sharding της MongoDB και την αποδοτικότητά του λόγω οριζόντιας κλιμάκωσης (horizontal scaling) βλέπουμε στα Κεφάλαια 2 και 3. Αυτό που μας ενδιαφέρει σε αυτήν την ανάλυση είναι ότι ο κατανεμημένος τρόπος αναζήτησης σε cluster της MongoDB είναι πιο αποδοτικός και γρήγορος από την αναζήτηση σε ένα μόνο shard, λόγω των εύστοχων ερωτημάτων που δρομολογεί ο query router – δηλαδή το κατάλληλο ερώτημα για το shard που έχει την απάντηση. Βέβαια, όπως παρατηρούμε, κάτι τέτοιο δεν τηρείται επακριβώς σε αυτό το αίτημα και στο επόμενο. Αυτό οφείλεται πιθανώς σε συμφόρηση δικτύου κατά την μέτρηση των χρόνων ή στο γεγονός ότι τα μηχανήματα είναι εικονικά και όχι πραγματικά. Στη συνέχεια, αναφέρουμε τα υπόλοιπα αιτήματα πελάτη, των οποίων μετρήσαμε τη χρονική απόκριση της πλατφόρμας. Οι εξηγήσεις και αναλύσεις είναι παρόμοιες με αυτό το αίτημα όπως και ο σκοπός διεξαγωγής τους.

Simple Business

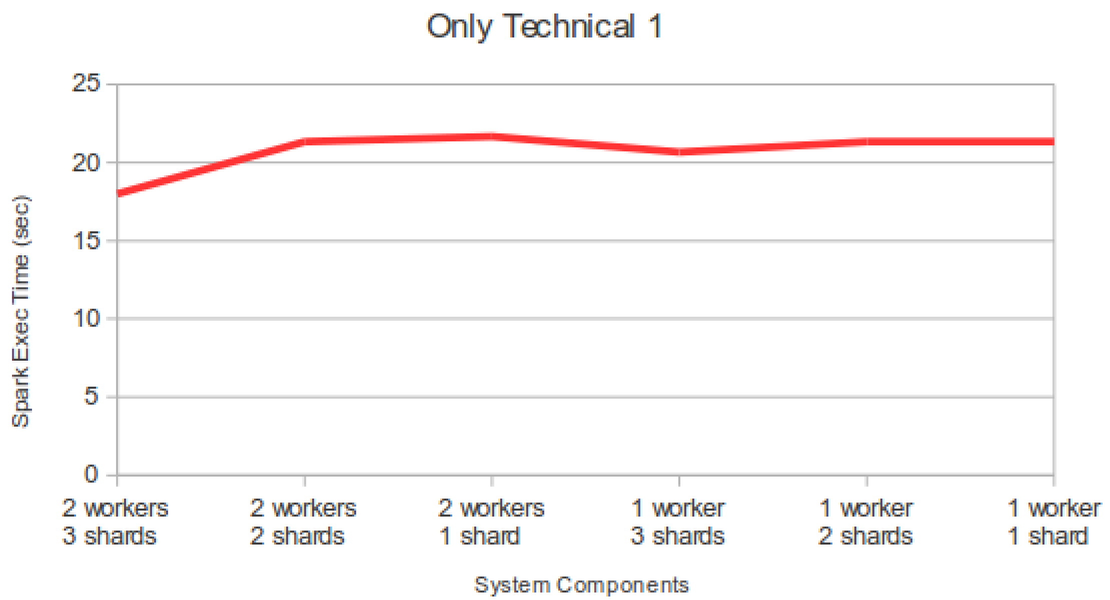


Πίνακας 7: Simple Business Query

Το παρόν αίτημα πελάτη ονομάζεται "Simple Business" και αποτελείται από τις λέξεις-κλειδιά "reliability". Ο πελάτης ζητάει από το σύστημα να του επιστρέψει τα καλύτερα προϊόντα που

προσφέρουν αξιοπιστία (reliability). Υπενθυμίζουμε ότι το σύστημα είναι προγραμματισμένο, εφόσον βρει τα προϊόντα που πρέπει, να προσφέρει στον πελάτη πρώτα τις φθηνότερες επιλογές ανά κατηγορία. Κατά τη διάρκεια επεξεργασίας του παραπάνω αιτήματος του πελάτη το σύστημα βρίσκει αυτά τα προϊόντα και τα επιστρέφει σε αυτόν. Στην τρέχουσα περίπτωση δεν υπάρχει ανάγκη για επίλυση αλληλοσυσχετίσεων καθώς δεν εμπλέκεται κάποιο τεχνικό χαρακτηριστικό στο αρχικό αίτημα.

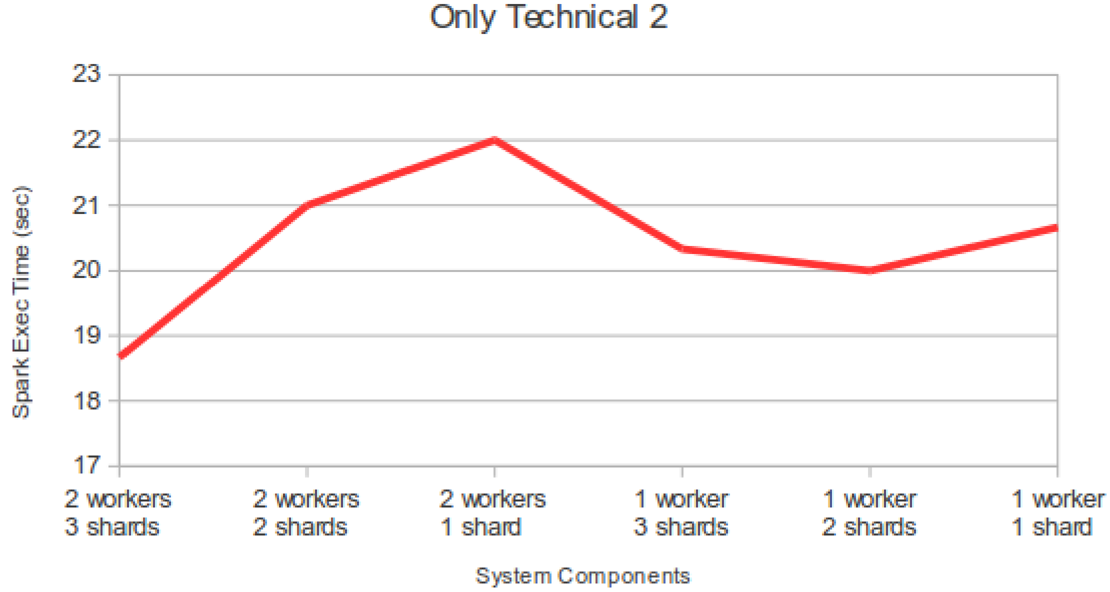
Only Technical 1



Πίνακας 8: Only Technical 1 Query

Αυτό το αίτημα πελάτη ονομάζεται "Only Technical 1" και αποτελείται από τις λέξεις-κλειδιά "journaling, snapshot". Ο πελάτης ζητάει από το σύστημά μας να του επιστρέψει τα προϊόντα που εμπεριέχουν τα τεχνικά χαρακτηριστικά journaling και snapshot μαζί. Το σύστημα βρίσκει αυτά τα προϊόντα, επιλύει τις αλληλοσυσχετίσεις (interrelationships) – δηλαδή, σε αυτήν την περίπτωση, βρίσκει τα business χαρακτηριστικά που επηρεάζονται θετικά ή αρνητικά από τα παραπάνω τεχνικά, κρατώντας όσα μένουν θετικά μετά από τις προσθαφαιρέσεις της επίλυσης – και επιστρέφει στον χρήστη πρώτα, προϊόντα που περιέχουν τα δύο τεχνικά χαρακτηριστικά που ζητήθηκαν, και σε δεύτερο χρόνο προϊόντα που περιέχουν τα business που υπολόγισε παραπάνω. Σχετικά με τις προσθαφαιρέσεις μπορούμε να δούμε στην παράγραφο "Interrelationship Scala Object" του Κεφαλαίου 4.

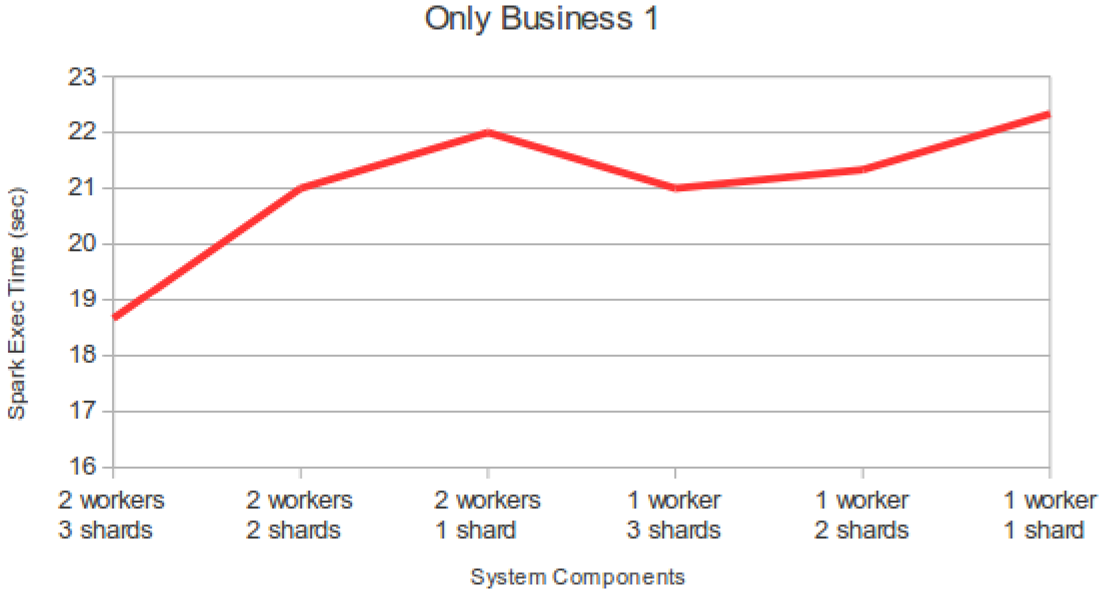
Only Technical 2



Πίνακας 9: Only Technical 2 Query

Το παρόν αίτημα πελάτη ονομάζεται "Only Technical 2" και αποτελείται από τις λέξεις-κλειδιά "snapshot, distributed, RAID, replication". Ο πελάτης ζητάει από το σύστημα να του επιστρέψει τα προϊόντα που προσφέρουν ταυτόχρονα snapshot, RAID, replication και είναι κατανεμημένα (distributed). Στη συνέχεια, το σύστημα βρίσκει αυτά τα προϊόντα, επιλύει τις αλληλοσυσχετίσεις (interrelationships) – δηλαδή, στην προκειμένη περίπτωση, εντοπίζει τα business χαρακτηριστικά που μπορεί να έχει ένα σύστημα όταν εμπεριέχει ταυτόχρονα τα παραπάνω τεχνικά χαρακτηριστικά, χρησιμοποιώντας τους κανόνες των αλληλοσυσχετίσεων – και, τέλος, επιστρέφει στον πελάτη τα προϊόντα με την σωστή σειρά. Όπως περιγράφουμε παραπάνω, η σειρά αυτή περιέχει πρώτα τα προϊόντα που ζήτησε άμεσα ο πελάτης, δηλαδή με τα τέσσερα technical, και στη συνέχεια τα προϊόντα μέσω αλληλοσυσχετίσεων που προτείνει το σύστημα.

Only Business 1



Πίνακας 10: Only Business 1 Query

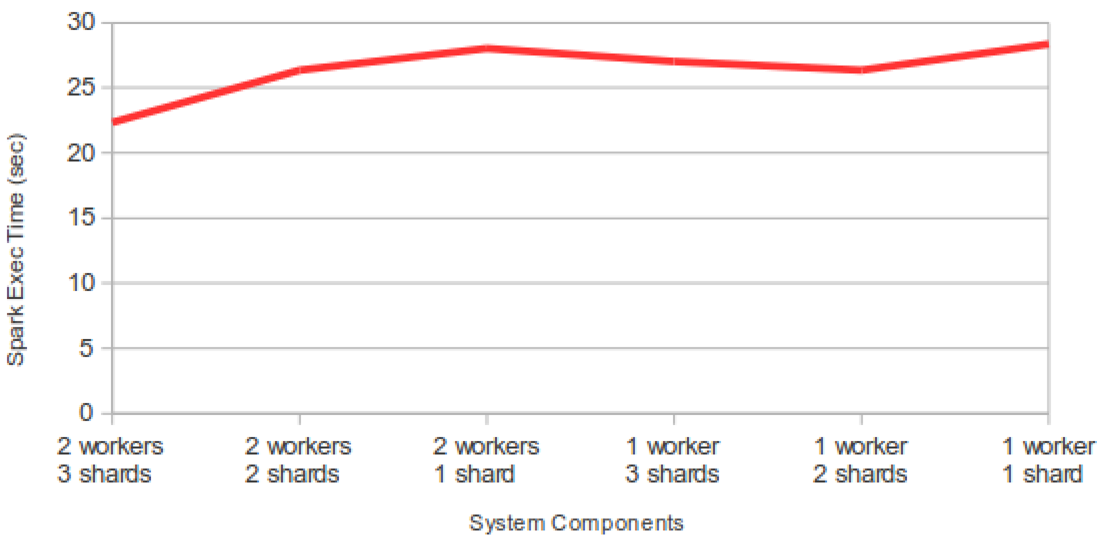
Αυτό το αίτημα πελάτη ονομάζεται "Only Business 1" και αποτελείται από τις λέξεις-κλειδιά "performance, high_availability". Ο πελάτης ζητάει από το σύστημά μας να του επιστρέψει τα προϊόντα που εμπεριέχουν τα business χαρακτηριστικά απόδοση (performance) και υψηλή διαθεσιμότητα (high availability) μαζί. Στη συνέχεια, το σύστημα βρίσκει αυτά τα προϊόντα και, εφόσον δεν υπάρχουν αλληλοσυσχετίσεις να επιλυθούν λόγω της απουσίας technical, τα επιστρέφει στον πελάτη.

Only Business 2

Πίνακας 11: Only Business 2 Query

Το παρόν αίτημα πελάτη ονομάζεται "Only Business 2" και αποτελείται από τις λέξεις-κλειδιά "reliability, recovery, security, storage_optimization". Εδώ, ο πελάτης ζητάει από το σύστημα να του επιστρέψει τα προϊόντα που προσφέρουν ταυτόχρονα αξιοπιστία (reliability), δυνατότητα ανάκτησης αρχείων (recovery), ασφάλεια (security) και βελτιστοποίηση αποθηκευτικού χώρου (storage optimization). Στη συνέχεια, το σύστημα βρίσκει αυτά τα προϊόντα και τα επιστρέφει στον χρήστη. Σε αυτό το αίτημα, όπως και στο προηγούμενο δεν υπάρχουν technical ώστε να λειτουργήσουν οι αλληλοσυσχετίσεις ανάμεσα στα χαρακτηριστικά. Στα επόμενα αιτήματα, θα συνυπάρχουν τεχνικά και business χαρακτηριστικά και θα προκαλείται η επίλυση αλληλοσυσχετίσεων.

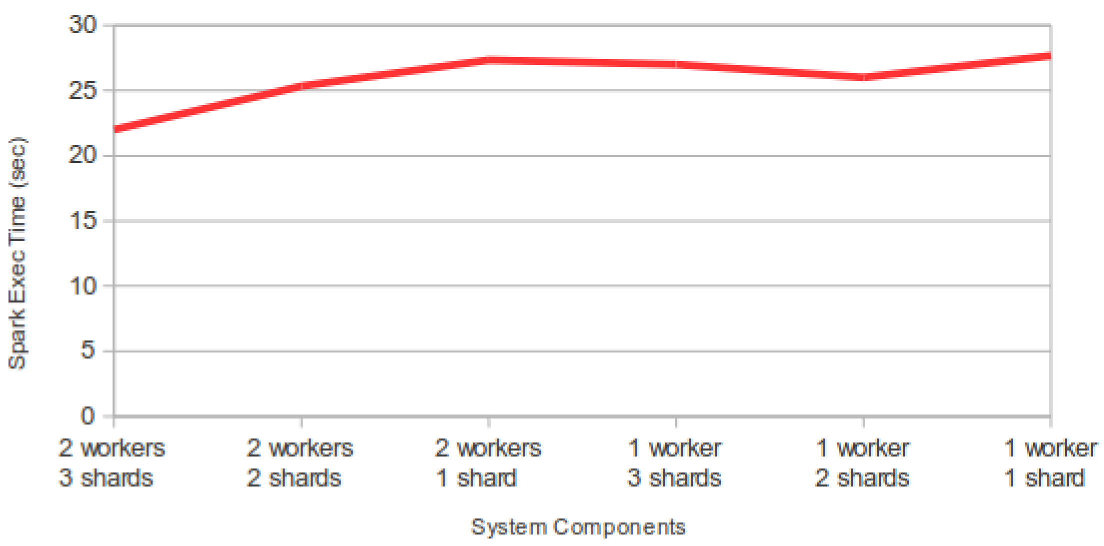
Technical & Business without conflict



Πίνακας 12: Technical & Business without conflict Query

Αυτό το αίτημα πελάτη ονομάζεται "Technical & Business without conflict" και αποτελείται από τις λέξεις-κλειδιά "compression, reliability". Ο πελάτης ζητάει από το σύστημά μας να του επιστρέψει τα προϊόντα που εμπεριέχουν ταυτόχρονα συμπίεση αρχείων αποθηκευτικού χώρου (compression) και αξιοπιστία (reliability). Το σύστημα βρίσκει πρώτα τα προϊόντα με το τεχνικό χαρακτηριστικό "compression" και στη συνέχεια επιλύει τις αλληλοσυσχετίσεις (interrelationships) που προκαλεί αυτό το χαρακτηριστικό. Σε αυτή τη φάση, συμβαίνει το ίδιο που περιγράφουμε στην επίλυση των αλληλοσυσχετίσεων παραπάνω, ενώ είναι επιπλέον πιθανό το τεχνικό χαρακτηριστικό που ζητείται (compression) να επηρεάζει αρνητικά το business που ζητείται (reliability). Κάτι τέτοιο θα μπορούσε να προκαλέσει μέχρι και την ακύρωσή της από το αίτημα, εφόσον οι συντελεστές προσθαφαίρεσης υπολόγιζαν την reliability μη θετική. Στην προκειμένη περίπτωση δεν συμβαίνει κάτι τέτοιο και το σύστημα επιστρέφει τα προϊόντα με τη σωστή σειρά, σαν να είχαμε ταυτόχρονα τα αιτήματα "Simple Technical" και "Simple Business".

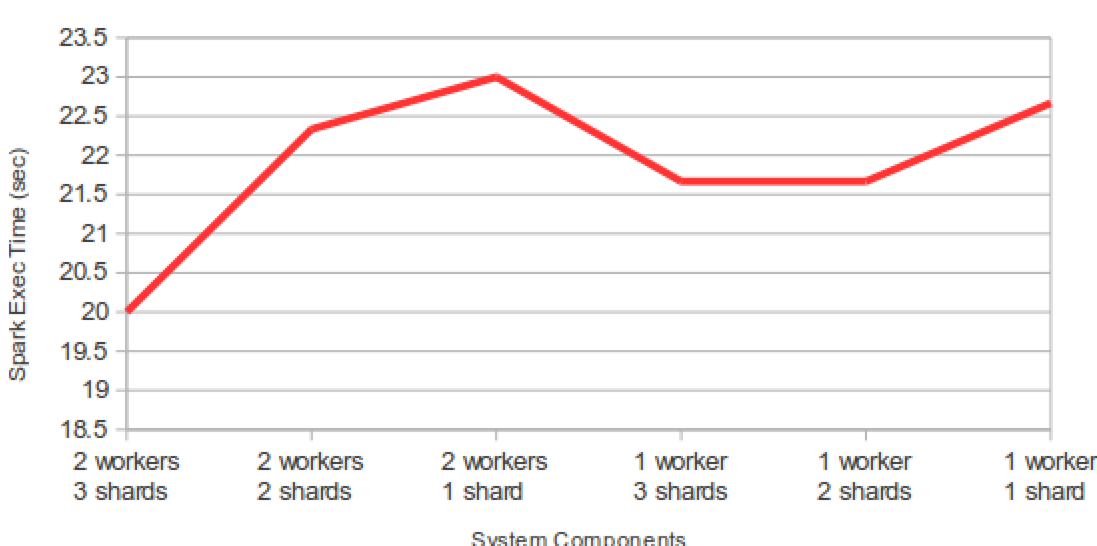
Technical & Business with conflict 1



Πίνακας 13: Technical & Business with conflict 1 Query

Το παρόν αίτημα πελάτη ονομάζεται "Technical & Business with conflict 1" και αποτελείται από τις λέξεις-κλειδιά "b+tree, storage_optimization". Ο πελάτης ζητάει από το σύστημα να του επιστρέψει τα προϊόντα που προσφέρουν ταυτόχρονα B^+ tree και βελτιστοποίηση αποθηκευτικού χώρου (storage optimization). Το σύστημα βρίσκει πρώτα τα προϊόντα που εμπεριέχουν B^+ tree και στη συνέχεια επιλύει τις αλληλοσυσχετίσεις (interrelationships). Εκτός από τα business χαρακτηριστικά που επηρεάζει θετικά, το B^+ tree επηρεάζει αρνητικά την βελτιστοποίηση αποθηκευτικού χώρου. Εάν οι συντελεστές προσθαφαίρεσης επηρεάζουν αρνητικά περισσότερο της μονάδας – η ύπαρξη στο αίτημα δίνει +1 στο business χαρακτηριστικό – τότε το business χαρακτηριστικό αφαιρείται από το αίτημα και κατ' επέκταση από την αναζήτηση προτεινόμενων προϊόντων. Αυτό συμβαίνει λόγω των κριτηρίων που εφαρμόζονται στο σύστημά μας και καθιστούν τα τεχνικά χαρακτηριστικά σημαντικότερα από τα business. Αναλυτικά, φαίνονται στο τέλος του Κεφαλαίου 3. Σε διαφορετική περίπτωση υπολογισμού, το σύστημα επιστρέφει μαζί με τα προϊόντα που υποστηρίζουν B^+ tree και προϊόντα που υποστηρίζουν βελτιστοποίηση αποθηκευτικού χώρου.

Technical & Business with conflict 2



Πίνακας 14: Technical & Business with conflict 2 Query

Αυτό το αίτημα πελάτη ονομάζεται "Technical & Business with conflict 2" και αποτελείται από τις λέξεις-κλειδιά "encryption, replication, storage_optimization, high_availability". Ο πελάτης ζητάει από το σύστημα να του επιστρέψει τα προϊόντα που υποστηρίζουν ταυτόχρονα κρυπτογράφηση αποθηκευτικού χώρου (encryption), replication, βελτιστοποίηση αποθηκευτικού χώρου (storage optimization) και υψηλή διαθεσιμότητα (high availability). Το σύστημα βρίσκει πρώτα τα προϊόντα που υποστηρίζουν ταυτόχρονα τα δύο τεχνικά χαρακτηριστικά (encryption και replication) και στη συνέχεια επιλύει τις αλληλοσυσχετίσεις (interrelationships) που αυτά προκαλούν. Με την ίδια λογική του προηγούμενου αιτήματος, το σύστημα καταλήγει ότι το storage optimization επηρεάζεται αρνητικά από το replication, ενώ η υψηλή διαθεσιμότητα επηρεάζεται θετικά από το replication και αρνητικά από το encryption. Εδώ, μπορούμε να δούμε ακόμα καλύτερα τη σημασία των συντελεστών προσθαφαίρεσης. Ανάλογα με τις τιμές τους διαμορφώνεται το αίτημα του πελάτη. Για παράδειγμα, εάν όλοι οι συντελεστές είναι σε μέτρο ίσοι με τη μονάδα, τότε το αίτημα γίνεται "encryption, replication, high_availability", καθώς μόνο η βελτιστοποίηση αποθηκευτικού χώρου εξουδετερώνεται από το τεχνικό χαρακτηριστικό "replication" σύμφωνα πάντα με τα κριτήρια που έχουμε βασίσει τη λύση του προβλήματός μας και φαίνονται αναλυτικά στο τέλος του Κεφαλαίου 3. Τέλος, το σύστημα επιστρέφει στον πελάτη τα προϊόντα με τη σωστή σειρά, όπως έχουμε αναλύσει σε προηγούμενα αιτήματα.

Παραπάνω είδαμε διαφορετικά αιτήματα που μπορεί ένας πελάτης να υποβάλει στην υπηρεσία μας. Είδαμε ότι χρόνοι απόκρισης σε ένα κατανεμημένο σύστημα είναι καλύτεροι από ότι σε ένα

σύστημα που έχει λιγότερα ή και ένα μοναδικό εικονικό μηχάνημα. Στην περίπτωση του Spark, είδαμε ότι όταν τα tasks μοιράζονται σε περισσότερα του ενός μηχανήματα, η εφαρμογή εκτελείται σε λιγότερο χρόνο, αφού κάθε μηχάνημα αναλαμβάνει να εκτελέσει ένα κλάσμα του φόρτου εργασίας. Ακόμη, στην περίπτωση της MongoDB, ισχύει κάτι αντίστοιχο για το Sharded Cluster που προσφέρει, αυτή τη φορά με την έννοια ότι η οριζόντια κλιμάκωση εδώ βοηθάει στις πιο εύστοχες αναζητήσεις μεταξύ των shards. Βέβαια, σε αυτό το Κεφάλαιο, η εστίαση και η ανάλυση για τις χρονικές αποκρίσεις έγινε μόνο στο πρώτο αίτημα πελάτη, και πρέπει να τονίσουμε ότι τα ίδια ισχύουν και για τα επόμενα αιτήματα που παρουσιάστηκαν.

Τα παραπάνω είδη αιτημάτων πελάτη επιλέχθηκαν προς περιγραφή και ανάλυση για να τονιστούν δύο κύρια κομμάτια αυτής της υπηρεσίας. Και τα δύο αναφέρθηκαν πολλές φορές κατά την παραπάνω ανάλυση. Το ένα αφορά, φυσικά, τον βασικό πυλώνα του συστήματος και δεν είναι άλλο από την επίλυση αλληλοσυσχετίσεων μεταξύ file system χαρακτηριστικών. Οι αλληλοσυσχετίσεις (interrelationships) και η λογική πίσω από αυτές – τα τεχνικά είναι σημαντικότερα και επηρεάζουν τα business – είναι και η βάση αυτής της διπλωματικής εργασίας. Το δεύτερο κομμάτι αφορά την χρονική απόκριση στις πλατφόρμες των Spark και MongoDB όταν γίνεται χρήση συμπλέγματος. Το κάθε cluster, είτε Spark είτε MongoDB, επωφελείται από την ιδιότητα της οριζόντιας κλιμάκωσης και προσφέρει υψηλές αποδόσεις – περισσότερη ανάλυση της οριζόντιας κλιμάκωσης γίνεται στο Κεφάλαιο 2.

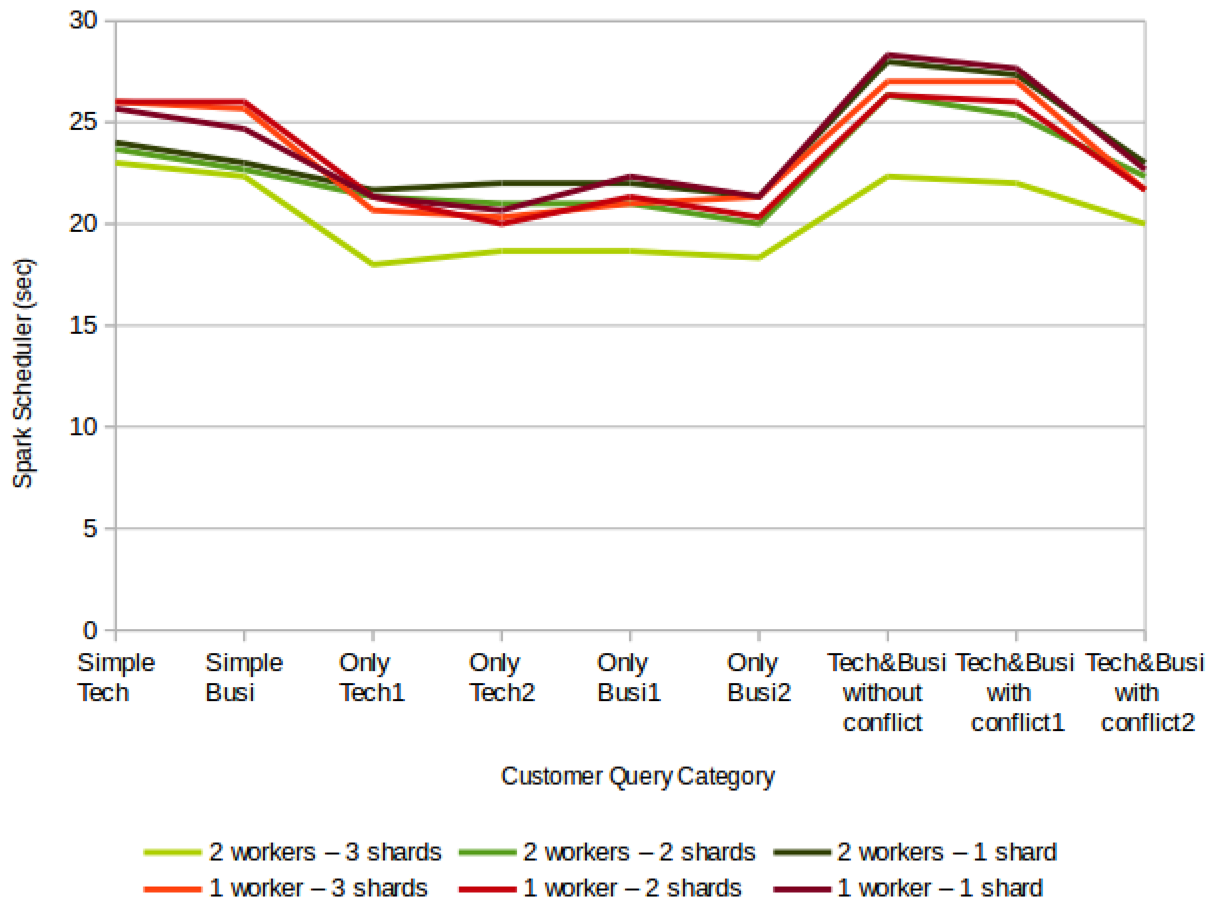
6

Εισαγωγή

Σε αυτό το Κεφάλαιο συγκεντρώνουμε καλύτερα τα αποτελέσματα της παρούσας εργασίας και αναφέρουμε συγκεκριμένα μέρη της που μπορούν πιθανόν να επαναχρησιμοποιηθούν από άλλες εφαρμογές. Παρουσιάζουμε αποτελέσματα και συμπεράσματα για την πλατφόρμα που δημιουργήσαμε και αναφέρουμε περιπτώσεις χρήσης κομματιών ή ακόμη και ολόκληρων πακέτων της και κλείνουμε με σχόλια για την έκβαση του πειράματος της εργασίας.

Συγκριτικά Αποτελέσματα

Στο προηγούμενο κεφάλαιο, παραθέσαμε διαγράμματα με αποτελέσματα πειραμάτων τα οποία είχαν ως αντικείμενα μελέτης τούς χρόνους απόκρισης της υπηρεσίας σε διαφορετικά αιτήματα πελάτη, και ταυτόχρονα σε διαφορετικές μορφοποιήσεις των components του συστήματος. Ως χρόνους απόκρισης μετράμε ουσιαστικά τον χρόνο – μέτρηση που παίρνουμε από τον Standalone Scheduler– σε δευτερόλεπτα που χρειάζεται το Spark για να φέρει εις πέρας την υποβληθείσα εφαρμογή μας με διαφορετική είσοδο κάθε φορά. Εφαρμόσαμε εννέα διαφορετικές εισόδους ως αιτήματα πελάτη σε dataset 100.000 προϊόντων αλλάζοντας κάθε φορά τις μορφοποιήσεις του Spark και της MongoDB. Το Spark δοκιμάστηκε με έναν και δύο κόμβους-σκλάβους (workers) – ο master τρέχει πάντα ως συντονιστής επιπλέον– και η MongoDB δοκιμάστηκε σε ένα Sharded Cluster των τριών, δύο και ενός shards. Παρακάτω, βλέπουμε μια συνολική εικόνα των μετρήσεων και διακρίνουμε στο ίδιο πλαίσιο τις διαφορετικές μορφοποιήσεις των components.



Πίνακας 15: Συγκριτικό Διάγραμμα των Queries

Μέσω αυτού του συγκριτικού διαγράμματος, βλέπουμε καλύτερα τα πλεονεκτήματα σε ταχύτητα και απόδοση ενός κατανεμημένου συστήματος σε σύμπλεγμα έναντι συστημάτων με λιγότερα μηχανήματα.

Παρακάτω, παραθέτουμε τους πίνακες των μετρήσεων.

ID		
1	Simple Technical	cow
2	Simple Business	reliability
3	Only Technical 1	journaling, snapshot
4	Only Technical 2	snapshot, distributed, RAID, replication
5	Only Business 1	performance, high availability
6	Only Business 2	reliability, recovery, security, storage optimization
7	Tech&Busi without conflict	compression, reliability
8	Tech&Busi w/ conflict 1	b+tree, storage optimization
9	Tech&Busi w/ conflict 2	encryption, replication, storage optimization, high availability

MongoDB\Spark	ID	2 workers				1 worker			
3 shards	1	23	23	23	23	26	24	27	27
	2	22.3	23	22	22	25.7	26	26	25
	3	18	18	18	18	20.7	20	21	21
	4	18.7	19	19	18	20.3	20	20	21
	5	18.7	18	18	20	21	21	21	21
	6	18.3	19	18	18	21.3	21	21	22
	7	22.3	23	22	22	27	27	27	27
	8	22	21	23	22	27	27	26	28
	9	20	20	20	20	21.7	22	21	22
2 shards	1	23.7	23	24	24	26	26	25	27
	2	22.7	23	23	22	26	26	26	26
	3	21.3	22	20	22	21.3	22	21	21
	4	21	21	20	22	20	20	20	20
	5	21	20	22	21	21.3	22	21	21
	6	20	19	21	20	20.3	19	21	21
	7	26.3	26	27	26	26.3	25	27	27
	8	25.3	26	25	25	26	25	26	27
	9	22.3	23	23	21	21.7	22	21	22
1 shard	1	24	25	23	24	25.7	27	25	25
	2	23	23	23	23	24.7	22	26	26
	3	21.7	22	22	21	21.3	23	19	22
	4	22	21	21	24	20.7	21	21	20
	5	22	22	21	23	22.3	23	23	21
	6	21.3	21	23	20	21.3	22	20	22
	7	28	27	28	29	28.3	30	27	28
	8	27.3	27	29	26	27.7	29	28	26
	9	23	24	23	22	22.7	23	24	21

Πίνακας 16 και Πίνακας 17: Πίνακες των μετρήσεων

Σε αυτό το σημείο, πρέπει να τονίσουμε ότι τα μηχανήματα στα οποία πήραμε τις μετρήσεις σε αυτήν την εργασία είναι εικονικά και όχι πραγματικά μηχανήματα, γι' αυτό και ορισμένες από τις μετρήσεις μπορεί να μην εμπεριέχουν απόλυτη ακρίβεια πραγματικών συνθηκών. Εντούτοις, τα εικονικά μηχανήματα αυτά μάς βοήθησαν να χρησιμοποιήσουμε όλα αυτά τα components που ενώσαμε, και μας επέτρεψαν να έχουμε μια γενική εικόνα συστήματος όσο γίνεται πιο κοντά στο πραγματικό.

Τα κατανεμημένα συστήματα των Spark και MongoDB που χρησιμοποιούμε βασίζονται στην ιδιότητα της οριζόντιας κλιμάκωσης. Σε ένα cluster υπολογιστικό σύστημα, το κάθε μηχάνημα λαμβάνει κάποιες εργασίες και λειτουργίες προς εκτέλεση. Το κλειδί εδώ είναι ότι λόγω της οριζόντιας κλιμάκωσης, στο καθέ μηχάνημα ανατίθεται ένα κλάσμα του φόρτου εργασίας, το οποίο δεν το επιβαρύνει, όσο πολύπλοκες και χρονοβόρες αποδειχθούν αυτές οι λειτουργίες. Με αυτή τη λογική, για το ίδιο φόρτο εργασίας, όσο περισσότερα μηχανήματα προσθέσουμε σε ένα cluster,

τόσο πιο γρήγορα και αποδοτικά θα ανταποκριθεί στο workload. Επομένως, σε περίπτωση που κάποιος ήθελε να προτείνει την κάθετη κλιμάκωση (vertical scaling) ως λύση στο πρόβλημα του μεγάλου φόρτου εργασίας – πράγμα που σημαίνει την αγορά ακριβού εξοπλισμού για ένα μόνο μηχάνημα, ώστε να γίνει ισχυρότερο, με περισσότερη RAM, CPUs, κλπ.– τότε εμείς θα ισχυριζόμασταν ότι υπάρχει τεράστιος περιορισμός στις μέρες μας για το πόσο εξειδικευμένο ακριβό hardware μπορεί να υποστηρίξει ένας μόνο server. Και σε περίπτωση που γίνει αρκετά ισχυρός ώστε να το υποστηρίξει, αυτό μπορεί να μεγαλώσει ανά πάσα στιγμή και εκθετικά, όπως και συμβαίνει στις μέρες μας με τα Big Data. Από την άλλη, δεν υπάρχει περιορισμός για το πόσα μηχανήματα μπορούν να συνδεθούν και να συναποτελούν ένα cluster. Ήδη εταιρίες που είναι μεγαθήρια στο χώρο έχουν αναπτύξει και χρησιμοποιούν clusters που αποτελούνται από χιλιάδες κόμβους και εκτελούν εφαρμογές σε περιβάλλοντα οριζόντιας κλιμάκωσης. Εφόσον, βρισκόμαστε στην εποχή των Big Data και Cloud Computing, συμπεραίνουμε ότι η οριζόντια κλιμάκωση είναι μονόδρομος.

Στη συνέχεια του κεφαλαίου, βλέπουμε πιθανά κομμάτια του συστήματος τα οποία μπορούν να επαναχρησιμοποιηθούν και σε ποιες περιπτώσεις εμφανίζονται κατάλληλα.

Επαναχρησιμοποίηση

Κατ' αρχάς, ο τρόπος επίλυσης αλληλοσυσχετίσεων (interrelationship resolution) μεταξύ των file system χαρακτηριστικών και η λογική που έχει στηθεί παρουσιάζει ελαστικότητα και προσαρμοσιμότητα. Τα business χαρακτηριστικά είναι πιο αφηρημένα σαν έννοιες και μπορούν να γίνουν κατανοητά από μεγαλύτερο ποσοστό χρηστών, ενώ τα technical εμφανίζονται πιο συγκεκριμένα και εξειδικευμένα και την ίδια στιγμή απαιτούν από τον χρήστη να έχει την κατάλληλη τεχνογνωσία πάνω σε αυτά ώστε να παίρνει σωστές σχετικές αποφάσεις. Στις μέρες μας, με την ίδια λογική χρησιμοποιείται ο διαχωρισμός technical και business του Ralph Kimball – βλ. Παράρτημα 1– για τα metadata των επιχειρήσεων. Τα τεχνικά metadata μιας επιχείρησης σχετίζονται με την εσωτερική δομή της και απαιτούν εξουσιοδότηση και εξειδίκευση για να γίνουν εφικτές η πρόσβαση και η κατανόησή τους. Από την άλλη, τα business metadata αφορούν έννοιες κατανοητές από μεγαλύτερο ποσοστό ατόμων, όπως κανόνες της επιχείρησης, ετήσιες αναφορές πωλήσεων, πολιτικές, οργανόγραμμα κ.ά. Βλέπουμε δηλαδή ότι τα business metadata μιας επιχείρησης είναι καταλληλότερα για την επικοινωνία μεταξύ των ατόμων. Με τη λογική αυτή, η παρούσα ανάλυση μπορεί να χρησιμοποιηθεί από επιχειρήσεις που έχουν ανάγκη την αλληλοσυσχέτιση τεχνικών και business χαρακτηριστικών χρησιμοποιώντας το σύστημα προσθαφαίρεσης συντελεστών. Ένα παράδειγμα τέτοιας άμεσης αλληλοεξάρτησης είναι το τεχνικό χαρακτηριστικό "ανάγκη για ανάπτυξη εφαρμογών σε Apache Spark" να επηρεάζει θετικά (αύξηση) το business χαρακτηριστικό "νέες θέσεις εργασίας".

Ακόμη, είναι φανερή η χρησιμότητα του συντακτικού αναλυτή που αναπτύξαμε. Το πρότυπο TOSCA Simple Profile in YAML χρησιμοποιείται κατά κόρον στο σύγχρονο Cloud Computing. Στην παρούσα εργασία ο συντακτικός αναλυτής κατασκευάστηκε για τη μετατροπή σε JSON, ειδικών προϊόντων Filesystem As A Service τα οποία ακολουθούν το πρότυπο TOSCA Simple Profile in YAML v1.1. Επομένως, κάθε πάροχος με ειδικά τροποποιημένα προϊόντα σύμφωνα με το πρότυπο της TOSCA – τα οποία έχουν τύπο που επεκτείνει κάποιον κόμβο της TOSCA – μπορούν να μετατρέπονται σε JSON μορφή μέσω του αναλυτής μας με ελάχιστες τροποποιήσεις αυτού (οι τροποποιήσεις αφορούν μόνο την αλλαγή ονομάτων του παραπάνω τύπου και των πεδίων του κόμβου).

Έκβαση του πειράματος

Τέλος, κάνουμε μια αναφορά στην έκβαση του πειράματος της παρούσας διπλωματικής εργασίας. Κληθήκαμε να υλοποιήσουμε ένα σύστημα που θα εξυπηρετεί αιτήματα πελάτη του τύπου «ικανοποίηση file system attributes» (χαρακτηριστικά αποθηκευτικού χώρου) σε περιβάλλον Cloud Computing και Big Data με κύρια εστίαση στις πιθανές αλληλοσυσχετίσεις μεταξύ των χαρακτηριστικών αυτών. Η υλοποίησή του στέφθηκε με επιτυχία ικανοποιώντας όλες τις απαιτήσεις. Η μελέτη και ανάλυση αλληλοσυσχετίσεων μεταξύ χαρακτηριστικών του ίδιου file system, μας βοήθησε να έχουμε μια γενικότερη και σαφέστερη εικόνα για την ελεύθερη αγορά προϊόντων τέτοιου είδους.

7

Εισαγωγή

Στο παρόν Κεφάλαιο περιγράφουμε τρεις σημαντικές έννοιες που βασίζονται στη χρήση του συστήματος που κατασκευάσαμε, ως λογική και ως σύνολο, και κυρίως στοχεύουν στη διεύρυνσή του. Γίνεται λόγος για το πιθανό μελλοντικό έργο βασιζόμενο πάνω στο παρόν έργο και για τις δυνατές επεκτάσεις και βελτιστοποιήσεις που μπορεί να υποστηρίξει η υπηρεσία. Με την ίδια σειρά παρουσιάζονται παρακάτω αυτές οι έννοιες.

Μελλοντικό Έργο

Σε αυτή τη φάση αναλύουμε ένα πιθανό μελλοντικό έργο το οποίο μπορεί να βασιστεί στο ήδη υπάρχον. Στην παρούσα εργασία, είδαμε μια πλατφόρμα η οποία προτείνει προϊόντα σε έναν πελάτη ανάλογα με το αίτημά του. Το σύστημα επεξεργάζεται το αίτημα του πελάτη βασιζόμενο στο ίδιο το αίτημα αυτό καθ' αυτό. Την επόμενη φορά που ο ίδιος πελάτης κάνει ένα νέο αίτημα, η επεξεργασία αυτού ακολουθεί την ίδια λογική, χωρίς να επηρεάζεται από το προηγούμενο αίτημα. Προτείνουμε, επομένως, να καταχωρούνται τα αιτήματα, ή ακόμα και τα σημαντικότερα προϊόντα που προτάθηκαν σε έναν πελάτη – αποδοτικά, για παράδειγμα μόνο το κλειδί της βάσης δεδομένων –, ώστε οι βέλτιστες απαντήσεις νέων αιτημάτων του ίδιου πελάτη να συνοδεύονται με προϊόντα που αποτελούν βέλτιστες απαντήσεις προηγούμενων αιτημάτων. Συγκεκριμένα, θα μπορούσε να έχει κάθε πελάτης ένα προφίλ με το οποίο συνδέεται στο σύστημα ώστε να αναγνωρίζεται η ταυτότητά του. Έπειτα, ο πελάτης θα διατύπωνε το αίτημά του, έστω αίτημα "N", και το σύστημα θα του επέστρεφε τα προϊόντα που ικανοποιούν καλύτερα αυτό το αίτημα. Ακόμη, έχοντας διαβάσει ήδη το προφίλ του χρήστη από τη στιγμή της σύνδεσης – το οποίο περιλαμβάνει μια λίστα από τα σημαντικότερα προϊόντα προηγούμενων αιτημάτων – το σύστημα θα συνοδεύει την απάντηση του αιτήματος "N" με έναν αριθμό προϊόντων από αυτή τη λίστα. Τα κριτήρια επιλογής από τη λίστα δεν είναι αυστηρά και γι' αυτό μπορεί να επιλέγει τυχαία. Είναι προφανές ότι σε κάθε καινούργιο πελάτη δεν θα εφαρμόζεται άμεσα αυτή η τεχνολογία, καθώς θα πρέπει να αποθηκευτεί πρώτα πληροφορία παλιών αιτημάτων που θα κάνει προσεχώς. Χρήστες με μεγαλύτερο ιστορικό χρήσης της υπηρεσίας θα απολαμβάνουν περισσότερο αυτήν την τεχνολογία.

Τα παραπάνω αποτελούν την πρόταση για το μελλοντικό έργο που θα βασιστεί στο υπάρχον σύστημα. Στη συνέχεια θα δούμε πιθανές επεκτάσεις και βελτιστοποιήσεις της εφαρμογής.

Επεκτάσεις και Βελτιστοποιήσεις

Οι επεκτάσεις που μπορούν να γίνουν πάνω στο συγκεκριμένο έργο ξεχωρίζονται σε επεκτάσεις τροποποίησης και επεκτάσεις μορφοποίησης. Οι μεν σχετίζονται με τροποποιήσεις που επεμβαίνουν ουσιαστικά στην παρούσα εφαρμογή, αλλάζοντας μέχρι και τους σκοπούς του ίδιου του συστήματος – πράγμα που οφείλεται στην ελαστικότητα του κώδικα για επαναχρησιμοποίηση, ενώ οι δε σχετίζονται περισσότερο με την εμφάνιση και τη λειτουργία του συστήματος σε γραφικό περιβάλλον φιλικό προς τον πελάτη. Παρακάτω, αναφέρουμε παραδείγματα και από τις δύο κατηγορίες.

Το σύστημα σαν οντότητα υποστηρίζει αλλαγές που μπορούν να λύσουν ακόμα και διαφορετικά προβλήματα. Για παράδειγμα, ο κώδικας που γράφτηκε παρουσιάζει απόλυτη ελαστικότητα στην αλλαγή των προτεραιοτήτων της σημαντικότητας ενός προϊόντος. Όπως είδαμε από το Κεφάλαιο 4, αλλάζοντας τις προτεραιότητες για τα προϊόντα που επιστρέφονται, γίνεται δυνατόν, μια διαφορετική θεωρία για τη σημαντικότητα των προϊόντων, που λύνει ένα διαφορετικό πρόβλημα, να στηθεί και να υλοποιηθεί πάνω στον υπάρχων κώδικα. Γενικότερα, η συγκεκριμένη λογική και χρήση του Spark accumulator, μπορεί να επεκταθεί ακόμη περισσότερο, όχι απαραίτητα για προϊόντα αυτή τη φορά, αλλά για άλλου είδους έγγραφα της MongoDB – νέα έγγραφα διαφορετικών δεδομένων. Έχοντας την ίδια λογική και αλλάζοντας τις συνθήκες μέσα στη `foreach`, μπορούμε να χρησιμοποιήσουμε τον accumulator και τα ζεύγη με τις προτεραιότητες για να ξεχωρίσουμε τη σειρά που ικανοποιούνται οι νέες συνθήκες από τα παραπάνω έγγραφα. Οι επόμενες επεκτάσεις που αναφέρουμε αφορούν μορφοποιήσεις και βελτιστοποιήσεις στην υπάρχουσα εφαρμογή.

Μια βασική βελτιστοποίηση του συστήματος είναι ο τρόπος εμφάνισης. Μπορεί να αναπτυχθεί γραφικό περιβάλλον που θα είναι φιλικότερο στο χρήστη με radio buttons, dropdown λίστες και checkboxes. Ο χρήστης μέσω του γραφικού περιβάλλοντος θα μπορεί να υποβάλει το αίτημά του για τα χαρακτηριστικά ενός προϊόντος (file system storage). Στη συνέχεια, η επεξεργασία θα γίνεται στο παρασκήνιο με τον ίδιο τρόπο, και θα επιστρέφεται και θα τυπώνεται η λίστα από τα κατάλληλα προϊόντα στο γραφικό περιβάλλον (Graphical User Interface – GUI). Σε αυτήν την περίπτωση βέβαια, θα μπορούσαν να προστεθούν πληροφορίες για το κάθε file system, του τύπου: ανάλογα με το όνομά του να τυπώνεται και μια συγκεκριμένη εικόνα στην εκτύπωση στην οθόνη – δεν χρειάζεται η εικόνα να υπάρχει απαραίτητα στη βάση.

Ακόμη μία επέκταση που θα μπορούσε να γίνει αφορά τα χαρακτηριστικά των file system προϊόντων. Στο αρχείο YAML του παρόχου, υπάρχει ήδη στο πεδίο "os" του κάθε "server_" κόμβου

ενός αντίστοιχου "filesystem_", καταχωρημένο το λειτουργικό αυτού του filesystem. Στην παρούσα εργασία, βέβαια, δεν ασχοληθήκαμε ιδιαίτερα με αυτό το πεδίο, αλλά μια επέκταση της εργασίας θα


```
node_templates:
  server1:
   type: tosca.nodes.Compute
   capabilities:
     host:
      properties:
      # omitted, no interest here
     os:
      properties:
      # omitted, no interest here
     requirements:
…
```


μπορούσε να προσθέσει το διάβασμα του είδους του λειτουργικού συστήματος που υποστηρίζει το προϊόν Filesystem As A Service. Από το YAML στο JSON, και από το JSON στη MongoDB και μέσω του συστήματός μας στον τελικό πελάτη, η πληροφορία για το είδος του operating system (os) θα ενδιέφερε τον πελάτη.

Σε αυτό το σημείο κλείνουμε το κεφάλαιο των πιθανών επεκτάσεων και του μελλοντικού έργου σχετικών με το σύστημα που κατασκευάσαμε. Έγινε αναφορά στο πιθανό μελλοντικό έργο του συστήματος, που το εντοπίσαμε στο πεδίο των Collaborative Recommendation Systems. Ακόμη, είδαμε σε ποιου είδους προβλήματα θα ήταν χρήσιμη η υπηρεσία που παρουσιάσαμε. Τέλος, αναφέραμε κάποιες δυνατές επεκτάσεις του έργου καθώς και βελτιστοποιήσεις που θα βοηθούσαν και θα αναδείκνυαν ακόμη περισσότερο το έργο όπως η υλοποίηση γραφικού περιβάλλοντος.

Σε αυτό το σημείο ολοκληρώνεται η ανάλυση των Κεφαλαίων της εργασίας. Ακολουθούν αμέσως μετά τα Παραρτήματα που περιλαμβάνουν χρήσιμες ενδιάμεσες πληροφορίες και εξηγήσεις για τεχνολογίες της εργασίας, τον ίδιο τον κώδικα, configuration αρχεία και ρυθμίσεις, και tutorial και διαδικασία εγκατάστασης της πλατφόρμας. Τέλος, παρουσιάζεται η βιβλιογραφία.

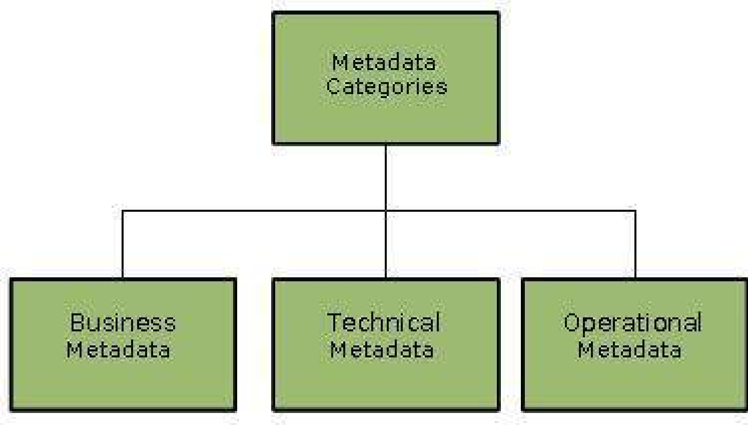
Παράρτημα 1

Οι τρεις πιθανές κατηγοριοποιήσεις των metadata

Οι Bretherton και Singley[4] ξεχώρισαν αυτό που λέμε μεταδεδομένα στις κατηγορίες structural/control και σε guide. Κατά αυτούς, στην πρώτη κατηγορία εμπίπτουν εκείνα που περιγράφουν αντικείμενα βάσεων δεδομένων, όπως columns, tables, indexes και keys, ενώ στην δεύτερη, όσα βοηθούν στην εύρεση συγκεκριμένων αντικειμένων, και τα οποία εκφράζονται συνήθως σε φυσική γλώσσα.

Από την άλλη, ο NISO (National Information Standards Organization) ξεχωρίζει τα μεταδεδομένα σε descriptive, structural και administrative.[5] Σύμφωνα με τον οργανισμό, τα descriptive metadata χρησιμοποιούνται για την ταυτοποίηση ενός αντικειμένου, βοηθώντας στην αναζήτηση και εύρεση του. Για παράδειγμα, σε ένα βιβλίο, ο τίτλος, η περίληψη, ο συγγραφέας, η θεματική ενότητα, ο εκδότης και οι λέξεις-κλειδιά. Στη συνέχεια, σύμφωνα πάλι με τον NISO, τα structural metadata έχουν να κάνουν με τον τρόπο που οργανώνονται τα μέρη ενός αντικειμένου. Στην περίπτωση του βιβλίου, ο τρόπος οργάνωσης σελίδων. Και τέλος για τα administrative, πρόκειται για όσα σχετίζονται με τον έλεγχο και τα δικαιώματα χρήσης της πληροφορίας.

Τέλος, η τελευταία κατηγοριοποίηση μεταδεδομένων είναι του Ralph Kimball, την οποία και χρησιμοποιήσαμε στην παρούσα εργασία. Ο Ralph Kimball ξεχωρίζει τα μεταδεδομένα σε technical και business. Ως technical (internal metadata) αναφέρονται όσα σχετίζονται με την εσωτερική δομή ενός συστήματος (για παράδειγμα DW/BI system), ενώ ως business (external metadata) όσα σχετίζονται αντίστοιχα με την εξωτερική (για DW/BI system, data availability, data interrelationships, η πηγή των data κλπ.). Στην πραγματικότητα, ο Ralph Kimball προσθέτει ακόμη μία κατηγορία, τα process metadata ή operational results metadata, που σχετίζονται με τα λειτουργικά αποτελέσματα ενός DW/BI system[7]. Εντούτοις, τα operational metadata δεν χρησιμοποιούνται στην παρούσα διπλωματική εργασία. Παρακάτω βλέπουμε τον διαχωρισμό του Ralph Kimball σχηματικά:



Εικόνα 33: Metadata Categories

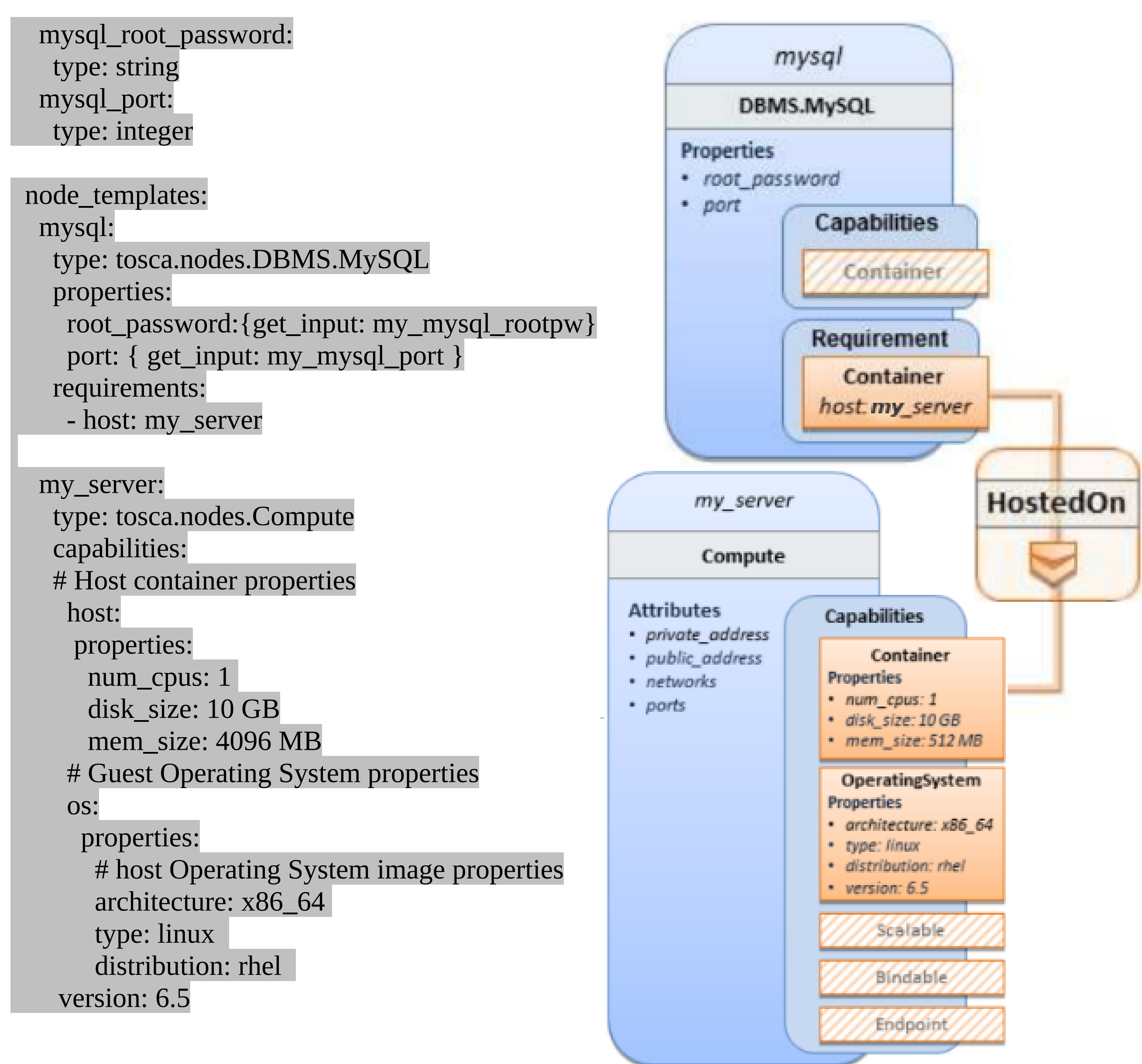
<u>TOSCA Simple Profile in YAML Version 1.1</u>

Η προδιαγραφή TOSCA Simple Profile in YAML Version 1.1 του οργανισμού OASIS, χρησιμοποιείται κατά κόρον στο Cloud Computing. Όπως βλέπουμε και στο Κεφάλαιο 3, το μοντέλο TOSCA χρησιμοποιεί προτυποποίηση για να περιγράψει φόρτο εργασιών υπηρεσιών νέφους (cloud workloads). Με τη χρήση προτύπων υπηρεσιών (service templates), τα cloud workloads περιγράφονται ως πρότυπες τοπολογίες (topology templates: γράφος με πρότυπους κόμβους που μοντελοποιούν τα στοιχεία του workload) και πρότυπες σχέσεις μεταξύ των κόμβων των τοπολογιών (relationship templates: οι ακμές του γράφου, δηλαδή η σύνδεση μεταξύ των κόμβων). Υπάρχουν κάποιοι ήδη ορισμένοι τύποι για κόμβους και σχέσεις που θεωρούνται γνωστοί, εντούτοις, είναι δυνατή η επέκταση αυτών, όπως πράξαμε στην εργασία. Οι γνωστοί κόμβοι είναι Compute, Root, ObjectStorage, BlockStorage, SoftwareComponentWebServer, Network, Port, WebApplicaton, DBMS, Database, Container κ.ά. και προσπελαύνονται με το URI "tosca.nodes.*" όταν χρειάζεται να ορίσουν νέο κόμβο (πεδίο "type" του νέου κόμβου). Ακόμη, οι γνωστές σχέσεις είναι Root, DependsOn, HostedOn, ConnectsTo, AttachesTo, RoutesTo, BindsTo, LinksTo κ.ά. και προσπελαύνονται με το URI "tosca.relationships.*", όταν χρειάζεται να ορίσουν νέα σχέση (πεδίο "derived_from:" της νέας σχέσης). Ένας TOSCA ενορχηστρωτής (orchestrator) διαβάζοντας το αρχείο YAML, επεξεργάζεται το πρότυπο υπηρεσίας (service template) και αναπτύσσει την πραγματική τοπολογία, με πραγματικούς κόμβους και συνδέσεις (σχέσεις) μεταξύ αυτών. Παρακάτω παραθέτουμε τροποποιημένο ένα παράδειγμα[10] από το επίσημο documentation του TOSCA Simple Profile YAMLv1.1 για να κατανοήσουμε καλύτερα και με μια ματιά όσα είπαμε μέχρι στιγμής:

Example 3 - Simple (MySQL) software installation on a TOSCA Compute node (Τροποποιημένο)

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for deploying a single server with MySQL software on top.


topology_template:   inputs:
```



```
  mysql_root_password:
    type: string
  mysql_port:
    type: integer

node_templates:
  mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      root_password:{get_input: my_mysql_rootpw}
      port: { get_input: my_mysql_port }
    requirements:
      - host: my_server

  my_server:
    type: tosca.nodes.Compute
    capabilities:
    # Host container properties
      host:
       properties:
         num_cpus: 1
         disk_size: 10 GB
         mem_size: 4096 MB
      # Guest Operating System properties
      os:
        properties:
          # host Operating System image properties
          architecture: x86_64
          type: linux
          distribution: rhel
     version: 6.5
```

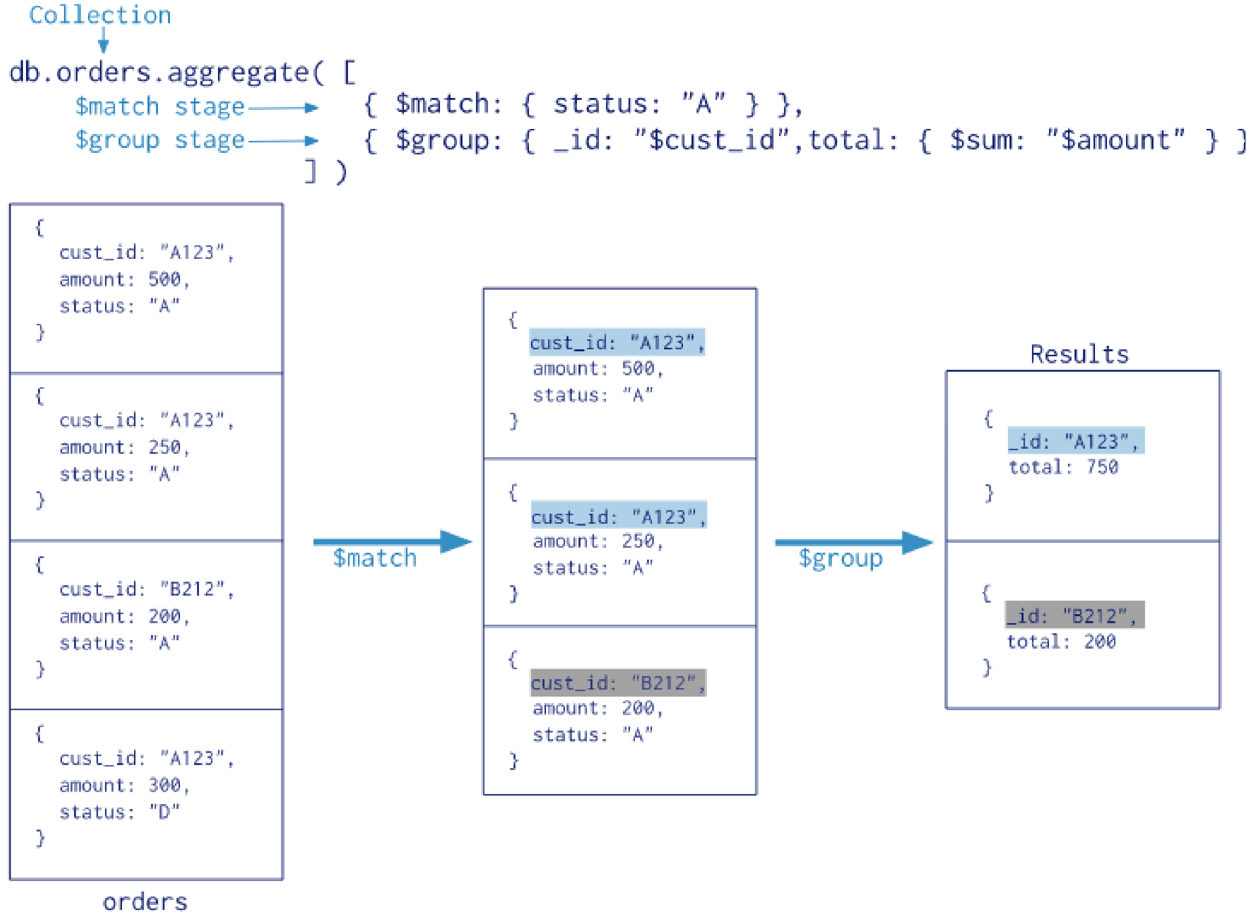

Εικόνα 34: Simple (MySQL) software installation on a TOSCA Compute node (Τροποποιημένο)

Στο παραπάνω παράδειγμα YAML αρχείου, έχουμε 2 νέους κόμβους (nodes) και μία σχέση (relationship) που τους συνδέει. Ο ένας κόμβος ονομάζεται "mysql" και είναι τύπου tosca.nodes.DBMS.MySQL, και ο άλλος "my_server" και τύπου tosca.nodes.Compute. Η σχέση που τους συνδέει εκφράζεται στα "requirements" του πρώτου κόμβου και δείχνει ότι ο "my_server" φιλοξενεί τον "mysql" (ο "mysql" είναι "HostedOn" "my_server"). Εδώ, βλέπουμε πολλά χρήσιμα σημεία της προδιαγραφής TOSCA Simple Profile YAMLv1.1. Κατ' αρχάς, τα "mysql", "HostedOn" και "my_server" είναι η πρότυπη τοπολογία (topology template), γράφος με 2 κόμβους και 1 ακμή. Ο κάθε κόμβος έχει τις δικές του ιδιότητες, με τον "mysql" να έχει κυρίως τιμές εισόδου, τιμές που καθορίζονται πλέον από τον ίδιο τον χρήστη και δεν παραμένουν σταθερές πάνω στην πρότυπη τοπολογία (όπως password, port, εδώ, ή και IP, username, κλπ), και τον "my_server" να έχει

δυνατότητες ("capabilities"), που, όπως βλέπουμε, περιγράφουν τις ιδιότητες του host και του λειτουργικού συστήματος ("os") που ο προγραμματιστής της εφαρμογής (application developer) πρέπει να λάβει υπόψιν του. Έτσι, όταν σηκωθούν οι πραγματικοί κόμβοι, καλό θα είναι ο προγραμματιστής να έχει προβλέψει, ώστε ο "my_server" κόμβος να έχει τον κατάλληλο αριθμό CPUs, χώρο στο δίσκο, μνήμη και λειτουργικό σύστημα ώστε να τρέξει επιτυχώς η εφαρμογή του.

<u>MongoDB Aggregation Pipeline</u>

Το aggregation pipeline της MongoDB αποτελεί ένα εργαλείο για συγκέντρωση και συνάρθροιση δεδομένων (data aggregation) που βασίζεται στην λογική των pipelines. Τα έγγραφα (documents) εισέρχονται σε ένα pipeline πολλαπλών σταδίων, το οποίο τα μετατρέπει σε συμπτυγμένα (aggregated) αποτελέσματα. Στην παράγραφο αυτή δίνουμε ένα σύντομο και περιεκτικό παράδειγμα για καλύτερη κατανόηση:



Εικόνα 35: Aggregation Pipeline

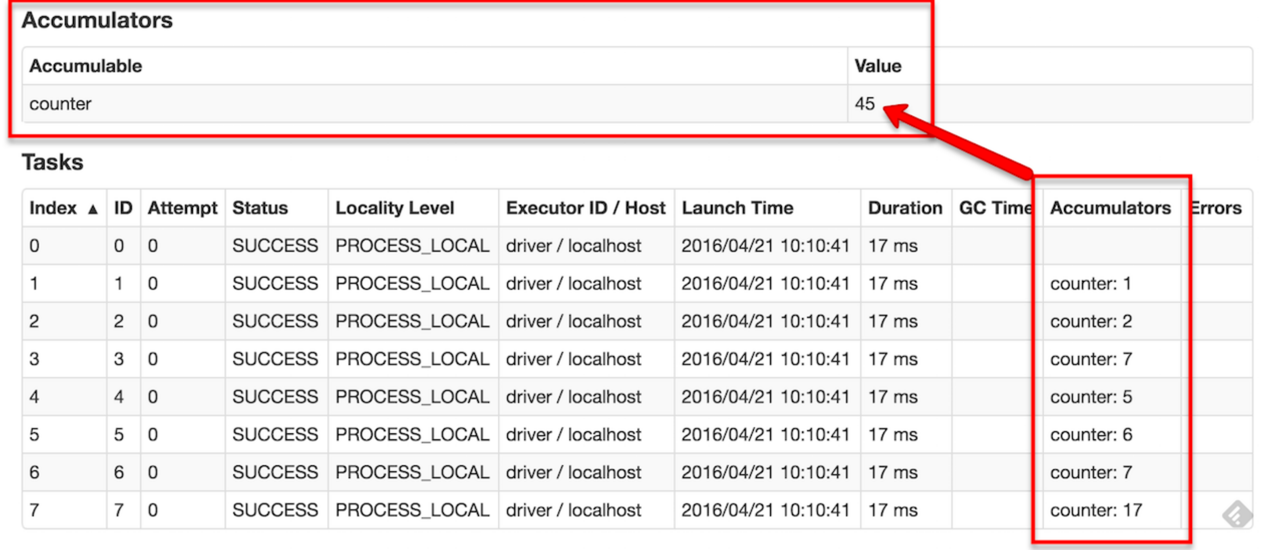
Το aggregation pipeline αποτελεί ένα εναλλακτικό μοντέλο του MapReduce, ενώ πολλές φορές μπορεί να δίνει την καλύτερη λύση μεταξύ των δύο. Εντούτοις, έχει περιορισμούς στους τύπους των τιμών και το μέγεθος των αποτελεσμάτων.[28]

Apache Spark AccumulatorV2

Υπό κανονικές συνθήκες, μια συνάρτηση που εκτελείται σε έναν απομακρυνσμένο κόμβο του Spark cluster (όπου περνιέται μέσω μιας Spark λειτουργίας τύπου map ή reduce), χρησιμοποιεί αντίγραφα για τις μεταβλητές της, τα οποία δεν είναι ίδια από κόμβο σε κόμβο. Αυτά τα αντίγραφα χρησιμοποιούνται αυστηρά στους κόμβους (workers) όπου ανήκουν, ενώ ούτε ο driver ούτε κάποιος άλλος κόμβος μαθαίνει για την τιμή τους. Το Spark, για να καλύψει την ανάγκη μια μεταβλητή να διατηρεί την ίδια τιμή ανάμεσα στους κόμβους του cluster, προσφέρει μία "shared variable" εν ονόματι accumulator.

Οι accumulators είναι μεταβλητές που δέχονται μόνο την πράξη της πρόσθεσης και χρησιμοποιούνται με υψηλές αποδόσεις στην παράλληλη επεξεργασία δεδομένων. Μπορούν να χρησιμοποιηθούν για την υλοποίηση μετρητών (counters), ενώ το Spark, όπως θα δούμε στη συνέχεια, υποστηρίζει accumulators για αριθμητικούς τύπους και ταυτόχρονα έχουμε τη δυνατότητα εμείς να προσθέσουμε δικούς μας τύπους accumulator, όπως και πράξαμε.

Υπάρχουν accumulators με ή χωρίς όνομα. Στην παρακάτω εικόνα, βλέπουμε έναν accumulator τύπου μετρητή, εν ονόματι counter. Η εικόνα προέρχεται από Web UI του Spark, και δείχνει αναλυτικά ποιο task και με ποιον τρόπο επηρεάζει τον accumulator.



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Εικόνα 36: Spark Accumulator

Με την ίδια λογική της πρόσθεσης πειραματιστήκαμε δημιουργώντας διαφορετικούς accumulators, για τους οποίους γίνεται αναφορά στη συνέχεια, για να καταλήξουμε στον "seqDocAccum", τον οποίο και χρησιμοποιήσαμε στην εφαρμογή μας. Ο seqDocAccum έχει τη λογική της λίστας. Αυτό που προστίθεται εδώ δεν είναι ακέραιοι αριθμοί, αλλά στοιχεία σε λίστα. Ακόμη, είναι απαραίτητο οι πράξεις στους accumulators να γίνονται στα πλαίσια κάποιας δράσης (action), επειδή σε διαφορετική περίπτωση – δηλαδή transformation – λόγω lazy evaluation δεν εκτελούνται οι πράξεις και ο accumulator δεν παίρνει την επιθυμητή τιμή. Παρακάτω βλέπουμε τη χρήση του longAccumulator(_), που είναι ένας από τους προεπιλεγμένους αριθμητικούς accumulators του Spark για την πρόσθεση αριθμών τύπου long:

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name:
Some(My Accumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
...

scala> accum.value
res2: Long = 10
```

Η πρόσθεση και εδώ, όπως και στους επόμενους accumulatos που κατασκευάσαμε, γίνεται παρόμοια με την προηγούμενη εικόνα. Κάθε διαφορετικό task της εκτέλεσης (τα tasks έχουν μοιραστεί στους workers του cluster) συμβάλλει με μία απομακρυνσμένα υπολογισμένη τιμή στην συνολική τιμή του accumulator. Στη συνέχεια βλέπουμε τους πειραματικούς accumulators που μας βοήθησαν στην κατασκευή του SeqDocAccum που χρησιμοποιούμε στην εφαρμογή μας:

- Ο AtomicLongAccumulator accumulator προσθέτει τιμές τύπου long,
- ο StringAccum χρησιμεύει ως string concatenator, αφού προσθέτει συμβολοσειρές (strings),
- ο DocAccum είναι δοκιμαστικός accumulator χωρίς κάποια ιδιαίτερη λειτουργία (απλά επιστρέφει έγγραφα),
- ο SeqDocOnly accumulator που επιστρέφει λίστα από έγγραφα (documents) και

- ο SeqDocAccum accumulator ο οποίος επιστρέφει ζεύγη τύπου (document,integer) και είναι κατάλληλος για την εφαρμογή μας

```
import java.util.concurrent.atomic.AtomicLong

case class AtomicLongAccumulator(initialValue: Long = 0) extends
AccumulatorV2[Long, Long] {
  private var _value = new AtomicLong(initialValue)
  override def value: Long = _value.get
  override def isZero: Boolean = value == 0
  override def copy(): AccumulatorV2[Long, Long] =
AtomicLongAccumulator(value)
  override def reset(): Unit = _value = new AtomicLong(0)
  override def add(v: Long): Unit = _value.addAndGet(v)
  override def merge(other: AccumulatorV2[Long, Long]): Unit =
add(other.value)
}
case class StringAccum(initialValue: String = "0") extends
AccumulatorV2[String, String] {
  private var _value = initialValue
  override def value: String = _value
  override def isZero: Boolean = value == "0"
  override def copy(): AccumulatorV2[String, String] = StringAccum(value)
  override def reset(): Unit = _value = "0"
  override def add(v: String): Unit = _value=_value+v
  override def merge(other: AccumulatorV2[String, String]): Unit =
add(other.value)
}
case class DocAccum() extends AccumulatorV2[Document,Document] {
  private var doc: Document = Document.parse("{}")
  override def value: Document = doc //.fold(Document.parse(""))
((x,y)⇒x+Document.parse("")+y)
  override def isZero: Boolean = value==Document.parse("{}")
  override def copy(): AccumulatorV2[Document,Document] = DocAccum()
  override def reset(): Unit = doc=Document.parse("{}")
  override def add(d: Document): Unit = doc=d //doc.append("AUU", d)
  override def merge(other: AccumulatorV2[Document,Document]): Unit =
add(other.value)
}
case class SeqDocOnly() extends AccumulatorV2[Seq[Document],Seq[Document]] {
  private var docs: Seq[Document] = Seq()
  override def value: Seq[Document] = docs
  override def isZero: Boolean = value==Seq()
  override def copy(): AccumulatorV2[Seq[Document],Seq[Document]] =
SeqDocAccum()
  override def reset(): Unit = docs=Seq()
  override def add(d: Seq[Document]): Unit = docs=docs.++:(d)
  override def merge(other: AccumulatorV2[Seq[Document],Seq[Document]]):
Unit = add(other.value)
}
case class SeqDocAccum() extends
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]] {
  private var docs: Seq[(Document,Int)] = Seq((Document.parse("{}"),0))
  override def value: Seq[(Document,Int)] = docs
  override def isZero: Boolean = value==Seq((Document.parse("{}"),0))
  override def copy():
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]] = SeqDocAccum()
  override def reset(): Unit = docs=Seq((Document.parse("{}"),0))
```

```
        override def add(d: Seq[(Document,Int)]): Unit = docs=docs.++:(d)
        override def merge(other: 
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]]): Unit = add(other.value)
    }
```

Τους παραπάνω accumulators πρέπει ακόμη να τους αρχικοποιήσουμε:

```
    val atomic = new AtomicLongAccumulator(0)
    val stringAccum = new StringAccum("0")
    val docAccum = new DocAccum()
    val SeqDocOnly = new SeqDocOnly()
    val seqDocAccum = new SeqDocAccum()
```

και να τους γνωστοποιήσουμε στο τρέχον SparkContext ως νέες shared μεταβλητές τύπου accumulator:

```
spark.sparkContext.register(atomic, "atomic")
spark.sparkContext.register(stringAccum, "stringAccum")
spark.sparkContext.register(docAccum, "docAccum")
spark.sparkContext.register(seqDocAccum, "seqDocOnly")
spark.sparkContext.register(seqDocAccum, "seqDocAccum")
```

.

Παράρτημα 2

Παράδειγμα JSON αρχείου με 3 filesystem προϊόντα

Παραδείγματα αρχείων TOSCA Simple Profile YAML version 1.1 φαίνονται στο Κεφάλαιο 3 και το Παράρτημα 1. Εδώ παραθέτουμε ένα παράδειγμα JSON αρχείου με 3 filesystem προϊόντα. Το παρόν JSON αρχείο αποτελεί μορφή εξόδου του αναλυτή μας.


```
[
   {  "name" : "HDFS",
      "service_type" : "FilesystemAsAService",
      "provider" : "Google",
      "price" : 830.0,
      "technical" : ["replication", "snapshot", "distributed"],
      "business" : ["performance", "high_availability", "reliability", 
"recovery", "load_balancing"]
   },
   {  "name" : "GlusterFS",
      "service_type" : "FilesystemAsAService",
      "provider" : "Google",
      "price" : 815.0,
      "technical" : ["replication", "snapshot", "distributed"],
```

```
        "business" : ["performance", "high_availability", "reliability",
"recovery", "load_balancing"]
    },
    {   "name" : "MooseFS",
        "service_type" : "FilesystemAsAService",
        "provider" : "Google",
        "price" : 560.0,
        "technical" : ["snapshot", "distributed"],
        "business" : ["reliability", "recovery", "high_availability",
"performance", "load_balancing"]
    }
]
```

Filesystem Generator:

Για δημιουργία YAML αρχείου ενός provider. Ο κάθε provider δίνει ένα αρχείο με τα Filesystem as a Service προϊόντα του. Καλείται με είσοδο τον αριθμό των προϊόντων που επιθυμούμε να δημιουργήσει. Μορφή κώδικα: bash script.

```
#!/bin/bash
productsToGenerate=$1

FilesystemName="HAMMER
BTRFS
ReFS
Ext4
ZFS
APFS
eCryptFS
RozoFS
XFS
MooseFS
GlusterFS
HDFS"

FilesystemPrice="110.0
80.0
130.0
70.0
150.0
230.0
70.0
75.0
60.0
190.0
195.0
210.0"

declare -A FilesystemAttr
#HAMMER
FilesystemAttr[0,0]="snapshot"
FilesystemAttr[0,1]="b+tree"
#BTRFS
```

```
FilesystemAttr[1,0]="journaling"
FilesystemAttr[1,1]="snapshot"
FilesystemAttr[1,2]="deduplication"
FilesystemAttr[1,3]="b-tree"
FilesystemAttr[1,4]="cow"
FilesystemAttr[1,5]="defragmentation"
FilesystemAttr[1,6]="checksum"
FilesystemAttr[1,7]="scrubbing"
FilesystemAttr[1,8]="stripping"
FilesystemAttr[1,9]="mirroring"
FilesystemAttr[1,10]="RAID"
FilesystemAttr[1,11]="replication"
FilesystemAttr[1,12]="compression"
#ReFS
FilesystemAttr[2,0]="deduplication"
FilesystemAttr[2,1]="scrubbing"
FilesystemAttr[2,2]="checksum"
FilesystemAttr[2,3]="b+tree"
#Ext4
FilesystemAttr[3,0]="journaling"
FilesystemAttr[3,1]="defragmentation"
FilesystemAttr[3,2]="encryption"
#ZFS
FilesystemAttr[4,0]="journaling"
FilesystemAttr[4,1]="snapshot"
FilesystemAttr[4,2]="defragmentation"
FilesystemAttr[4,3]="cow"
FilesystemAttr[4,4]="compression"
FilesystemAttr[4,5]="encryption"
FilesystemAttr[4,6]="deduplication"
FilesystemAttr[4,7]="replication"
FilesystemAttr[4,8]="RAID"
#APFS
FilesystemAttr[5,0]="snapshot"
FilesystemAttr[5,1]="encryption"
FilesystemAttr[5,2]="cow"
#eCryptFS
FilesystemAttr[6,0]="encryption"
#RozoFS
FilesystemAttr[7,0]="distributed"
FilesystemAttr[7,1]="erasure_coding"
#XFS
FilesystemAttr[8,0]="journaling"
FilesystemAttr[8,1]="defragmentation"
FilesystemAttr[8,2]="b+tree"
#MooseFS
FilesystemAttr[9,0]="distributed"
FilesystemAttr[9,1]="snapshot"
#GlusterFS
FilesystemAttr[10,0]="snapshot"
FilesystemAttr[10,1]="distributed"
FilesystemAttr[10,2]="replication"
#HDFS
FilesystemAttr[11,0]="distributed"
FilesystemAttr[11,1]="replication"
FilesystemAttr[11,2]="erasure_coding"
FilesystemAttr[11,3]="snapshot"

# no Business attributes required in input, only Technical
```

```
Tech="journaling
encryption
distributed
compression
erasure_coding
replication
cow
b-tree
b+tree
defragmentation
deduplication
snapshot
checksum
RAID
mirroring
scrubbing
stripping"

TechPrice="100.0
200.0
200.0
160.0
350.0
250.0
50.0
60.0
70.0
150.0
190.0
170.0
110.0
200.0
300.0
290.0
310.0"

fsName=($FilesystemName)
fsNameLen=${#fsName[*]}
fsPrice=($FilesystemPrice)
t=($Tech)
tLen=${#t[*]}
tPrice=($TechPrice)

echo -n "Generating "
echo -n $productsToGenerate
echo " products..."

max_attr=12 # max number of attributes on a single FS
for prod in `seq 1 $productsToGenerate`; do
  let "ind=$RANDOM % $fsNameLen"
  name=${fsName[$ind]}
  price=${fsPrice[$ind]}
  let "numOfAttr=$RANDOM % $max_attr+1"
  attrList="['${FilesystemAttr[$ind,0]}"
  i=0
  until [[ ${FilesystemAttr[$ind,0]} == ${t[$i]} ]]; do
    let i=i+1
  done
  price=$(tclsh <<< "puts [expr $price+${tPrice[$i]}]")
```

```
  j=1
  until [ "${FilesystemAttr[$ind,$j]}" ==  "" ] || [ $j -ge $numOfAttr ]; do
    attrList+="','${FilesystemAttr[$ind,$j]}"
    i=0
    until [[ ${FilesystemAttr[$ind,$j]} == ${t[$i]} ]];do
      let i=i+1
    done
    price=$(tclsh <<< "puts [expr $price+${tPrice[$i]}]")
    let j=j+1
  done
  attrList+="']"
  . ./getRandom.sh 1000000 # creates a random number between 1 and 1.000.000
  price=$(tclsh <<< "puts [expr $price+$R/1000000.0]")
  cat >> products.yamlGen << EOF
  filesystem$prod:
   type: FilesystemAsAService
   properties:
    size: 10 GB
    snapshot_id: { get_input: storage_snapshot_id$prod }
    product_name: $name
    product_price: $price
    attributes: $attrList
EOF
done
```

Έξοδος: αρχείο τύπου YAML, βλ. Κεφάλαιο 3.

Συντακτικός αναλυτής:

Δέχεται ως είσοδο ένα YAML version 1.1 αρχείο, έξοδος του παραπάνω κώδικα, διαβάζει τα Filesystem as a Service προϊόντα και τα τυπώνει σε ένα καινούργιο αρχείο τύπου JSON. Μορφή κώδικα: Scala Object.

```
import org.mongodb.scala._
import org.mongodb.scala.MongoClientSettings._
import scala.collection.JavaConverters._
import scala.io.Source._
import Interrelationships._
import org.yaml.snakeyaml.Yaml
import java.io._
import org.bson.Document


object InputProducts {

def main(args: Array[String]):Unit ={
    val yamlFile = new File("/home/kapsoulis/products.yaml")
    val iterStr = fromFile(yamlFile).getLines
    var input = """"""
    iterStr.foreach { line ⇒ input=input+"\n"+line}
    val yaml = new Yaml()
    val data = yaml.load(input)
               .asInstanceOf[java.util.LinkedHashMap[String,
java.util.LinkedHashMap[String, Object]]]
               .asScala
    var prod:Seq[Document] = Seq()
```

```
    val file = "products.json"
    val writer = new BufferedWriter(new FileWriter(file))
    writer.write("[")
    val tt = data.get("topology_template")
    tt match {
      case Some(t) ⇒
        val nt = t.get("node_templates")
        var l = nt.toString()
        l = l.substring(1,l.length-1)
        var list = l.split("=|\\, ").toList
        var i = 1
        while (list contains "filesystem"+s"${i}") {
          while (list(list.indexOf("filesystem"+s"${i}")+2)!
="FilesystemAsAService")
            list=list.drop(list.indexOf("filesystem"+s"${i}")+2)
          var ix = list.indexOf("filesystem"+s"${i}")+2
          var j = 12
          var attr:List[String] = List()
          if (list(ix+j)!="[]") {
            while (list(ix+j).substring(list(ix+j).length-3,list(ix+j).length)!
="]}}")
              j=j+1
            if (j==12)
              attr = attr.+:(list(ix+12).substring(1, list(ix+12).length-3))
            else if (j==13) {
              attr = attr.+:(list(ix+12).substring(1))
              attr = attr.+:(list(ix+13).substring(0,list(ix+j).length-3))
            } else {
              attr = attr.+:(list(ix+12).substring(1))
              (13 to (j-1)).map(a=>attr=attr.+:(list(ix+a)))
              attr = attr.+:(list(ix+j).substring(0,list(ix+j).length-3))
            }
          }
          // Generate the relative business attributes, through
Interrelationships.scala
          var busiAttr:List[String] = List()
          interr(attr).map(x⇒ if (x._2>0)
            busiAttr=busiAttr.++(List(x._1)))
          writer.write(s"""{"name": "${list(ix+8).toString()}", "service_type":
"${list(ix)}", "provider": "Amazon",
            |"price": ${list(ix+10).toDouble}, "technical": $
{"[\""+attr.mkString("\",\"")+"\"]"},
            |"business": ${"[\""+busiAttr.mkString("\",\"")
+"\"]"}}""".stripMargin.replaceAll("[\n]","")+",\n")
          i=i+1
        }
      case None ⇒ println("none")
    }
    writer.write("{}]")
    writer.close()
  }
}
```

Η εφαρμογή.

Κύριο πρόγραμμα με τη main συνάρτηση που συνδέεται με MongoDB και Apache Spark, ζητάει από το χρήστη είσοδο, την επεξεργάζεται και τυπώνει το επιθυμητό αποτέλεσμα. Μορφή κώδικα: Scala Object.

```
import org.bson.Document
import ClusterSparkContext._
import Interrelationships._
import com.mongodb.spark.MongoSpark
import scala.util.matching.Regex
import org.apache.spark.util.AccumulatorV2
import scala.io.StdIn
import com.mongodb.spark.config.ReadConfig

object ProductRecommender {
    def main(args: Array[String]):Unit ={
    def userAnswer(): (List[String],List[String]) = {
      var success = false
      var answer = ""
      var answerArray: List[String] = null
      while(!success) {
        try {
          println("\n* * * What would you like your service to have? * * *")
          answer = StdIn.readLine().toString()
          answerArray = answer.replaceAll("[^A-Za-z0-9 _+]", " ").split("
").toList
          success = true
        }catch {
           case exception: Any =>
             {println("""Wrong input, please demand wisely along technical and
business attributes...
                         a two-word attribute should be separated by "_".
""".trim.stripMargin)}
        }
      }
      // divide tech&busi
      var tech:List[String]=List()
      var busi:List[String]=List()
      answerArray.map(a⇒
        if (t contains a )
          tech=tech.+:(a)
        else if (b contains a)
          busi=busi.+:(a))
      return (tech,busi)
    }

    // User's choice
    val uAnswer = userAnswer()
    val userTech = uAnswer._1
    var userBusi = uAnswer._2
    val uI = interr(userTech)
    var userInterr:List[String] = List()
    uI.map(x ⇒ if (x._2>0) userInterr=userInterr.++(List(x._1)))
    // Check for conflicting interrelationships.
    var conflicting:List[String] = List()
    userInterr.foreach { busiAttr =>
```

```
      if (userBusi.contains(busiAttr)) {
        uI.foreach(x=>
          if(x._1==busiAttr)
            if (x._2<=0)
              conflicting=conflicting.++(List(busiAttr)))
      }
    }
    userBusi=userBusi.filterNot{conflicting.toSet}
    userInterr=userInterr.filterNot{conflicting.toSet}

    // Final output's initialization
    case class SeqDocAccum() extends
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]] {
      private var docs: Seq[(Document,Int)] = Seq((Document.parse("{}"),0))
      override def value: Seq[(Document,Int)] = docs
      override def isZero: Boolean = value==Seq((Document.parse("{}"),0))
      override def copy():
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]] = SeqDocAccum()
      override def reset(): Unit = docs=Seq((Document.parse("{}"),0))
      override def add(d: Seq[(Document,Int)]): Unit = docs=docs.++:(d)
      override def merge(other:
AccumulatorV2[Seq[(Document,Int)],Seq[(Document,Int)]]): Unit = add(other.value)
    }
    val seqDocAccum = new SeqDocAccum()
    val sc = ClusterSparkContext.spark.sparkContext
    sc.register(seqDocAccum, "seqDocAccum")

    /** Technical Resolution **/ /** Interrelationship Resolution **/ /**
Business Resolution **/

    /** CONNECT TO SPARK & RUN **/

    /** Load from SparkContext. **/
    val readConfig = ReadConfig(Map("partitioner" -> "MongoShardedPartitioner"),
Some(ReadConfig(sc)))
    var rdd = MongoSpark.load(sc)//,readConfig) // is another way
    // Ask MongoDB First - not to bring the whole database on one RDD: (BE LAZY)
    val tempTech = userTech.mkString("\", $eq:\"")
    val tempBusi = userBusi.mkString("\", $eq:\"")
    val tempInterr = userInterr.mkString("\", $eq:\"")
/** Technical Resolution in MongoDB before passing to Spark. **/
    val aggregatedRdd = rdd.withPipeline(Seq(Document.parse(
        s"""{ $$match: { $$or: [{ technical:{$$eq:"${tempTech}"} },
                                { business:{$$eq:"${tempBusi}"} },
                                { business:{$$eq:"${tempInterr}"} }] } }""")))
    // Then ask Spark
    aggregatedRdd.foreach{ doc ⇒
      var prodTech = doc.get("technical").toString()
      var prodBusi = doc.get("business").toString()
      var techBool = if (userTech.isEmpty) false
                     else userTech.forall{a⇒prodTech.contains(a)}
      var busiBool = if (userBusi.isEmpty) false
                     else userBusi.forall{a⇒prodBusi.contains(a)}
      var interrBool = if (userInterr.isEmpty) false
                     else userInterr.forall{a⇒prodBusi.contains(a)}
      if (techBool && busiBool) // ideal product!!
        seqDocAccum.add(Seq((doc,0)))
      else if (techBool) // if this product has ALL the technical the user wants
        seqDocAccum.add(Seq((doc,1)))
```

```
      else if (busiBool) // if this product has ALL the business the user wants
        seqDocAccum.add(Seq((doc,2)))
      else if (interrBool)
        seqDocAccum.add(Seq((doc,3))) // The tech that ur looking for, give also
these business.
      // Can add more. Obviously with less demanding.
    }
    println("\n\n* * * R e s u l t s * * *\n")
    seqDocAccum.value.filter(x ⇒ !x._1.isEmpty()).sortBy(s ⇒ (s._2,
s._1.getDouble("price")) ).take(25).foreach(x⇒
      if (!x._1.isEmpty())
        println(x._1.getString("name")+"\t"+
                x._1.getDouble("price")+"\t"+
                x._1.getString("provider")+"\t"+
                x._1.getString("service_type")+"\t"+
                x._1.get("technical")+
                x._1.get("business"))
    )
    sc.stop()
    }
}
```

Interrelationships:

Scala Object που χρησιμοποιείται για την επίλυση των αλληλοσυσχετίσεων μεταξύ technical και business χαρακτηριστικών. Χρησιμοποιεί τη συνάρτηση "`interr(technical:List[String]) : List[(String,Int)]`" η οποία παίρνει ως είσοδο τη λίστα των τεχνικών χαρακτηριστικών ενός filesystem και επιστρέφει ως έξοδο τη λίστα των business χαρακτηριστικών που αντιστοιχούν. Μορφή κώδικα: Scala Object.

```
object Interrelationships {

    val t = List("journaling", "encryption", "distributed", "compression",
"erasure_coding", "replication", "cow", "b-tree", "b+tree", "defragmentation",
"deduplication", "snapshot", "checksum", "RAID", "mirroring", "scrubbing",
"stripping")
    val b = List("performance", "reliability", "recovery", "security",
"high_availability", "storage_optimization", "load_balancing")
    // interrelationship(technical:List[String]) => business:List[String]
    def interr(technical:List[String]) : List[(String,Int)] = {

      var business:List[(String,Int)]=List()
      def relateWith(busiAttr:String, infl:Int) {
        val influ = infl
        if (business.exists(_._1==busiAttr)) {
          var v = business.filter(_._1==busiAttr)(0)._2
          business = business.updated(business.indexOf((busiAttr,v)),
(busiAttr,v+influ))
        } else {
          business = business.++(List((busiAttr,influ)))}
```

```
}

technical.foreach{tech ⇒ tech match {
    case "journaling" ⇒ {
      relateWith("performance",-1)
      relateWith("reliability", 1)
      relateWith("recovery", 1)}

    case "encryption" ⇒ {
      relateWith("performance", -1)
      relateWith("security", 1)
      relateWith("high_availability", -1)}

    case "distributed" ⇒ {
      relateWith("performance", 1)
      relateWith("high_availability", 1)
      relateWith("load_balancing", 1)}
      relateWith("storage_optimization", -1)

    case "compression" ⇒ {
      relateWith("performance", -1)
      relateWith("storage_optimization", 1)}

    case "erasure_coding" ⇒ {
      relateWith("performance", -1)
      relateWith("high_availability", 1)
      relateWith("storage_optimization", 1)}

    case "replication" ⇒ {
      relateWith("performance", 1)
      relateWith("high_availability", 1)
      relateWith("storage_optimization", -1)}

    case "cow" ⇒ {
      relateWith("performance", -1)
      relateWith("reliability", 1)}

    case "b-tree" ⇒ {
      relateWith("performance", 1)
      relateWith("storage_optimization", -1)}

    case "b+tree" ⇒ {
      relateWith("performance", 1)
      relateWith("storage_optimization", -1)}

    case "defragmentation" ⇒ {
      relateWith("performance", 1)
      relateWith("storage_optimization", 1)}

    case "deduplication" ⇒ {
      relateWith("reliability", 1)
      relateWith("storage_optimization", 1)}

    case "snapshot" ⇒ {
      relateWith("reliability", 1)
      relateWith("recovery", 1)
      relateWith("storage_optimization", 1)
      relateWith("high_availability", 1)}

    case "checksum" ⇒ {
```

```
          relateWith("reliability", 1)}

        case "RAID" ⇒ {
          relateWith("reliability", 1)}

        case "mirroring" ⇒ {
          relateWith("high_availability", 1)}

        case "scrubbing" ⇒ {
          relateWith("high_availability", 1)}

        case "stripping" ⇒ {
          relateWith("high_availability", 1)}

        case x ⇒ println // Extend: Add more interrelationships.
      }
    }

    var busi:List[String] = List()
    business.map(x⇒
      if (x._2>0)
        busi=busi.++(List(x._1)))
    return business
  }
  implicit def remove(l:List[(Any,Any)], i: Int) = {
    if (i>=l.size) l
    else l.take(i) ++ l.drop(i+1)
  }
}
```

Configuration του SparkContext

που επιβάλλει η εφαρμογή. Μορφή κώδικα: Scala Object.

```
import org.apache.spark.sql.SparkSession

object ClusterSparkContext {

    private val USERNAME = "vrettos"
    private val PASSWORD = "vrettos"
    private val SERVER = "147.102.19.151"
    private val PORT   = "27021"
    private val DATABASE = "recommender"
    private val COLLECTION = "products"

    // W/ SparkSession builder, unlike SparkConf,
    // we enter all the configuration in the form .config("SparkURI", "String").
    // Here we access SparkContext through "spark.SparkContext".

    val spark = SparkSession.builder()
    .master("spark://master:7077")
    .appName("Product Recommender System")
    .config("spark.jars", "/home/kapsoulis/ProductRecommenderSystem.jar")
    .config("spark.mongodb.input.partitioner", "MongoShardedPartitioner$")
```

```
    .config("spark.mongodb.input.partitionerOptions.shardKey", "price") // 
"shardKey: String" in MongoShardedPartitioner.scala, so
      // +Screenshots MongoShardedPartitioner.scala
   // +Anyway, price is already the best since its dependency to name & 
technical
      // +DOES NOT SUPPORT compound shard key: [name,price],(name,price),
{name:1,price:1}, name:1,price:1,{name,price},name||price, name,price
    .config("spark.mongodb.input.uri",s"mongodb://${USERNAME}:${PASSWORD}@$
{SERVER}:${PORT}/${DATABASE}.${COLLECTION}?authSource=admin")
    .config("spark.mongodb.output.uri",s"mongodb://${USERNAME}:${PASSWORD}@$
{SERVER}:${PORT}/${DATABASE}.${COLLECTION}?authSource=admin")
    .getOrCreate()
    // Gets an existing SparkSession or, if there is no existing one, creates a 
new one based on the options set in this builder.
    // This method first checks whether there is a valid thread-local 
SparkSession, and if yes, return that one. It then checks whether there is a 
valid global default SparkSession, and if yes, return that one. If no valid 
global default SparkSession exists, the method creates a new SparkSession and 
assigns the newly created SparkSession as the global default.
    // In case an existing SparkSession is returned, the config options 
specified in this builder will be applied to the existing SparkSession.
}
```

Παράρτημα 3

Για configuration αρχεία της MongoDB και του Apache Spark, ανατρέχουμε στο Παράρτημα 4.

Παρακάτω, παραθέτουμε συμπληρωματικό κώδικα που είναι απαραίτητος για την λειτουργία της πλατφόρμας, εκτός από τον κώδικα της Cosine Similarity που ανήκει στην πρώτη μεθοδολογία προσέγγισης του προβλήματος.

build.sbt:

Απαραίτητο αρχείο για την οργάνωση και μεταγλώττιση της εφαρμογής, καθώς και την εξαγωγή της σε ένα αρχείο JAR, το οποίο θα υποβληθεί στο Spark με την εντολή: "spark-submit --class ProductRecommender ProductRecommenderSystem.jar" (ο master καθορίζεται μέσα στην εφαρμογή, στο ClusterSparkContext.scala), και το οποίο περιλαμβάνει όλες τις απαραίτητες αλληλεξαρτήσεις βιβλιοθηκών για να τρέχει η εφαρμογή (library dependencies). Εκτελούμε με "`sbt package`" στον root folder του project, και το JAR δημιουργείται μέσα στον φάκελο `target/scala-2.11/` του root folder.

```
name := "File System Recommender"

version := "1.0"

fork in run           :=   true
connectInput in run   :=   true
scalaVersion := "2.11.7"

scalacOptions += "-deprecation"

libraryDependencies ++= Seq(
  "org.mongodb.spark" %% "mongo-spark-connector" % "2.1.0",
  "org.apache.spark" %% "spark-core" % "2.1.0",
  "org.apache.spark" %% "spark-sql" % "2.1.0",
  "org.mongodb.scala" %% "mongo-scala-driver" % "2.0.0",
  "org.yaml" % "snakeyaml" % "1.17")

assemblyJarName in assembly := "ProductRecommenderSystem.jar"
mainClass in assembly := some("ProductRecommender")
assemblyJarName := "ProductRecommender.jar"
val meta = """META.INF(.)*""".r

assemblyMergeStrategy in assembly := {
  case PathList("META-INF", xs @ _*) => MergeStrategy.discard
  case PathList("javax", "servlet", xs @ _*) => MergeStrategy.discard
  case PathList(ps @ _*) if ps.last endsWith ".html" => MergeStrategy.discard
  case n if n.endsWith(".conf") => MergeStrategy.discard
  case meta(_) => MergeStrategy.discard
  case x => MergeStrategy.first
}
assemblyExcludedJars in assembly := {
  val cp = (fullClasspath in assembly).value
  cp filter { f =>
    f.data.getName.contains("jackson") ||
    f.data.getName.contains("slf4j") ||
    f.data.getName.contains("avro-1.7.7") ||
    f.data.getName.contains("lz4") ||
    f.data.getName.contains("log4j") ||
    f.data.getName.contains("spark-tags") ||
    f.data.getName.contains("commons-compress") ||
    f.data.getName.contains("scala-") ||
    f.data.getName.contains("scala-xml") ||
    f.data.getName.contains("xz") ||
    f.data.getName.contains("paranamer") ||
    f.data.getName.contains("metrics-core") ||
    f.data.getName.contains("zookeeper") ||
    f.data.getName.contains("snappy-java") ||
    f.data.getName.contains("netty") ||
    f.data.getName.contains("hadoop")||
    f.data.getName.contains("parquet")||
    f.data.getName.contains("javax")||
    f.data.getName.contains("pyrolite")||
    f.data.getName.contains("commons")||
    f.data.getName.contains("avro")||
    f.data.getName.contains("curator")||
    f.data.getName.contains("aopalliance")||
    f.data.getName.contains("guava") ||
    f.data.getName.contains("compress-lzf-1.0.3")||
    f.data.getName.contains("curator-recipes")||
    f.data.getName.contains("guava-14.0.1")||
```

```
    f.data.getName.contains("unused-1.0.0")||
    f.data.getName.contains("jersey-guava-2.22.2")||
    f.data.getName.contains("jersey-common-2.22.2")||
    f.data.getName.contains("janino-3.0.0")||
    f.data.getName.contains("univocity-parsers-2.2.1")||
    f.data.getName.contains("chill_2.11-0.8.0")||
    f.data.getName.contains("jersey-server-2.22.2")||
    f.data.getName.contains("jersey-media-jaxb-2.22.2")||
    f.data.getName.contains("hk2-locator-2.4.0-b34")||
    f.data.getName.contains("unused-1.0.0")||
    f.data.getName.contains("stream-2.7.0")||
    f.data.getName.contains("jersey-container-servlet-2.22.2")||
    f.data.getName.contains("jersey-container-servlet-core-2.22.2")||
    f.data.getName.contains("osgi-resource-locator-1.0.1")||
    f.data.getName.contains("jetty-util-6.1.26")||
    f.data.getName.contains("ivy-2.4.0")||
    f.data.getName.contains("guice-3.0")||
    f.data.getName.contains("oro-2.0.8")||
    f.data.getName.contains("py4j-0.10.4")||
    f.data.getName.contains("cglib-2.2.1-v20090111")||
    f.data.getName.contains("jersey-client-2.22.2")||
    f.data.getName.contains("hk2-api-2.4.0-b34")||
    f.data.getName.contains("javassist-3.18.1-GA")||
    f.data.getName.contains("validation-api-1.1.0.Final")||
    f.data.getName.contains("metrics-graphite-3.1.2")||
    f.data.getName.contains("antlr4-runtime-4.5.3")||
    f.data.getName.contains("leveldbjni-all-1.8")||
    f.data.getName.contains("chill-java-0.8.0")||
    f.data.getName.contains("kryo-shaded-3.0.3")||
    f.data.getName.contains("commons")||
    f.data.getName.contains("jsr305-1.3.9")||
    f.data.getName.contains("minlog-1.3.0")||
    f.data.getName.contains("xmlenc-0.52")||
    f.data.getName.contains("objenesis-2.1")||
    f.data.getName.contains("xbean-asm5-shaded-4.4")||
    f.data.getName.contains("jets3t-0.7.1")||
    f.data.getName.contains("protobuf-java-2.5.0")||
    f.data.getName.contains("RoaringBitmap-0.5.11")
  }
}
```

getRandom.sh[23]:

bash script που μας δίνει έναν τυχαίο αριθμό μεταξύ του 1 και της εισόδου του. Maximum επιτρεπτή είσοδος 2^{60}–1. Χρήση κατά κόρον στον Filesystem Generator, βλ. παραπάνω.

```
#!/usr/bin/env bash
# Generates a random integer in a given range

# computes the ceiling of log2
# i.e., for parameter x returns the lowest integer l such that 2**l >= x
log2() {
  local x=$1 n=1 l=0
  while (( x>n && n>0 ))
  do
    let n*=2 l++
  done
```

```
  echo $l
}
# uses $RANDOM to generate an n-bit random bitstring uniformly at random
#  (if we assume $RANDOM is uniformly distributed)
# takes the length n of the bitstring as parameter, n can be up to 60 bits
get_n_rand_bits() {
  local n=$1 rnd=$RANDOM rnd_bitlen=15
  while (( rnd_bitlen < n ))
  do
    rnd=$(( rnd<<15|$RANDOM ))
    let rnd_bitlen+=15
  done
  echo $(( rnd>>(rnd_bitlen-n) ))
}
# alternative implementation of get_n_rand_bits:
# uses /dev/urandom to generate an n-bit random bitstring uniformly at random
#  (if we assume /dev/urandom is uniformly distributed)
# takes the length n of the bitstring as parameter, n can be up to 56 bits
get_n_rand_bits_alt() {
  local n=$1
  local nb_bytes=$(( (n+7)/8 ))
  local rnd=$(od --read-bytes=$nb_bytes --address-radix=n --format=uL
/dev/urandom | tr --delete " ")
  echo $(( rnd>>(nb_bytes*8-n) ))
}
# for parameter max, generates an integer in the range {0..max} uniformly at
random
# max can be an arbitrary integer, needs not be a power of 2
rand() {
  local rnd max=$1
  # get number of bits needed to represent $max
  local bitlen=$(log2 $((max+1)))
  while
    # could use get_n_rand_bits_alt instead if /dev/urandom is preferred over
$RANDOM
    rnd=$(get_n_rand_bits $bitlen)
    (( rnd > max ))
  do :
  done
  echo $rnd
}
# MAIN SCRIPT
# check number of parameters
if (( $# != 1 && $# != 2 ))
then
  cat <<EOF 1>&2
Usage: $(basename $0) [min] max

Returns an integer distributed uniformly at random in the range {min..max}
min defaults to 0
(max - min) can be up to 2**60-1
EOF
  exit 1
fi
# If we have one parameter, set min to 0 and max to $1
# If we have two parameters, set min to $1 and max to $2, if we want a random
between a range, i.e. [min,max].
max=0
while (( $# > 0 ))
do
```

```
  min=$max
  max=$1
  shift
done
# ensure that min <= max
if (( min > max ))
then
  echo "$(basename $0): error: min is greater than max" 1>&2
  exit 1
fi
# need absolute value of diff since min (and also max) may be negative
diff=$((max-min)) && diff=${diff#-}
# Results a.k.a. R
export R=$(( $(rand $diff) + min ))
```

Cosine Similarity:

Πρώτη μεθοδολογική προσέγγιση: μέθοδος cosine similarity[24]:

ComputeCosSim.scala

```
import org.bson.Document
import ClusterSparkContext._

object ComputeCosSim {
//   (p_i dot u)/||p_i|| ||up||, p_i=product_i and up=user's answer
  def computeCosSim (productDoc:Document, uAnswerDoc:Document,
userNorm:Double) : Double = {
    val pNorm:Double = computeNorm(productDoc)
    val uNorm:Double = userNorm
    def takeAsList (doc:Document):List[Double] = {
      return List(doc.getDouble("price"), doc.getInteger("journaling").toDouble,
                  doc.getInteger("snapshot").toDouble,
doc.getInteger("replication").toDouble)
    }
    val cosSim =  ((takeAsList(productDoc) zip takeAsList(uAnswerDoc))
             .map{Function.tupled(_ * _)}
             .sum)/(pNorm*uNorm)
    return cosSim
  }


  /** Computes the norm of a product. User's answer maybe inputed as product w/
zero values in the respect fields. **/
  def computeNorm (productDoc:Document):Double = {
    val allConsideredAttributes = List("journaling", "snapshot", "replication",
"price")
    val l:List[Double] = null
    val norm:Double = math.sqrt(
        (allConsideredAttributes.map {
            attr ⇒ if (attr!="price") {
                     productDoc.getInteger(attr).toDouble
                   }
                   else if (attr=="price")
productDoc.getDouble("price").toDouble
```

```
                else 0.0
        }).sum
      )
      return norm
    }
}
```

ProductRecommender.scala: Για την πρώτη μεθοδολογία προσέγγισης του προβλήματος, δίνεται ο κώδικας που αναπτύχθηκε.

```
import ComputeCosSim._
import org.mongodb.scala._
import org.bson.Document
import org.mongodb.scala.model.Aggregates._
import org.mongodb.scala.model.Accumulators._
import org.mongodb.scala.model.Filters._
import org.mongodb.scala.model.Projections._
import org.mongodb.scala.model.Sorts._
import org.mongodb.scala.model.Updates._
import Helpers._
import MongoFactory._
import LocalSparkContext._
import org.mongodb.scala.bson.conversions.Bson
import com.mongodb
import com.mongodb.spark.MongoSpark
import com.mongodb.spark.config._
import scala.util.matching.Regex
import scala.collection.immutable.ListMap

object ProductRecommender {

  def main(args: Array[String]) ={

    val maxProductsToRecommend = 10
    InputProducts
    def userAnswer(): List[String] = {
      var success = false
      var answer = ""
      var answerArray: List[String] = null
      while(!success) {
        try {
          println("What would you like your service to have?")
          answer = "snapshot, journaling"
          answerArray = answer.replaceAll("[^A-Za-z0-9 ]", "").split(" ").toList
          success = true
        }catch {
           case exception: Any => {println("""Wrong input, please demand wisely 
along name of service, service type,
                                   provider name, price, journaling, snapshot or 
replication...""".trim.stripMargin)}
        }
      }
      return answerArray
    val userInput: Seq[String]  = List("snapshot:1", "replication:0")
    var simArray: Array[Double] = null

    /** Load from SparkContext. **/
    val rdd = MongoSpark.load(spark.sparkContext)
```

```
    /** Cosine similarity between user's choice & data. **/
    val uAnswer = userAnswer()
    uAnswer.foreach {println}
    //uAnswerDoc is userAnswer to Doc
    val uAnswerDoc: Document = InputProducts.productToDoc(
        Product.apply("_", "filesystem", "_", 0, if (uAnswer contains
"journaling") 1; else 0,
        if (uAnswer contains "snapshot") 1; else 0, if (uAnswer contains
"replication") 1; else 0))

    val maxProd = maxProductsToRecommend
    var cosineSimList : List[(Document, Double)] = List()
    val uNorm: Double = ComputeCosSim.computeNorm(uAnswerDoc)
    rdd.map{productDoc =>(ComputeCosSim.computeCosSim(productDoc, uAnswerDoc,
uNorm),productDoc)}
       .sortBy(_._1,false)
       .collect()
       .foreach(println)

    //Cleanup
    println("\nDropping collection...")
    collection.drop().results()
    // Stop MongoDB Instance
    MongoFactory.mongoClient.close()
    // Stop Spark
    spark.stop()
  }
}
```

Παράρτημα 4

Αυτοματοποίηση σύνδεσης με ssh:

Στην παρούσα εργασία έχουμε διαθέσιμα 3 virtual machines που πρέπει να επικοινωνούν μεταξύ τους ανά πάσα στιγμή. Το ssh configuration είναι απαραίτητο ανάμεσα στα 3 εικονικά μηχανήματα λόγω χρήσης Spark (cluster) και MongoDB (sharding). Στην περίπτωση λειτουργίας του Spark, ο cluster manager χρειάζεται να συνδέεται στους workers για να μπορεί να σηκώνει, οργανώνει και να αναθέτει πόρους στους executors απομακρυνσμένα, και ο driver πρέπει να αναθέτει tasks σε αυτούς. Ακόμη, ο master πρέπει να έχει ssh πρόσβαση στους workers για να εκτελείται το Spark. Επίσης, για το sharding configuration της MongoDB, δεν αρκεί μόνο το keyfile για την επικοινωνία των shards, και το ssh configuration κρίνεται αναγκαίο.

Η παρακάτω απλή διαδικασία αυτοματοποίησης ssh σύνδεσης ανάμεσα σε υπολογιστές πρέπει να εφαρμόζεται σε ζεύγη μηχανημάτων. Στην περίπτωσή μας, έχουμε 3 εικονικά μηχανήματα, άρα 3 ζεύγη. Για κάθε ζεύγος ακολουθούμε την παρακάτω διαδικασία:

1. Εάν δεν υπάρχει εγκατάσταση ssh, τότε εγκαθιστούμε και στα δύο μηχανήματα (1 ζεύγος) με τις εντολές:

```
sudo apt install openssh-client
sudo apt install openssh-server
```

Ελέγχουμε την δημιουργία του κρυφού φακέλου `.ssh/` στην θέση `~/`.

2. Δημιουργούμε δημόσιο κλειδί και στα δύο μηχανήματα, ώστε να γίνεται σύνδεση από κάθε μεριά:

```
ssh-keygen -t rsa
```

3. Σε κάθε μηχάνημα εκτελούμε την εντολή σύνδεσης με ssh, αντικαθιστώντας το "ssh" με "ssh-copy-id" για να αποθηκευτεί ο κωδικός πρόσβασης του άλλου μηχανήματος:

```
ssh-copy-id kapsoulis@147.102.19.151
```

Στη συνέχεια, μας ζητείται να βάλουμε τον κωδικό σύνδεσης με ssh για τελευταία φορά. Εδώ, προσπαθούμε να συνδεθούμε στο μηχάνημα 147.102.19.151 ως χρήστης "kapsoulis".

<u>Εγκατάσταση Apache Spark from scratch</u>**:**

Η εγκατάσταση του Apache Spark προϋποθέτει την εγκατάσταση Java και Scala εκ των προτέρων στο εκάστοτε υπολογιστή, ενώ πρέπει να εφαρμόζεται σε όλα τα μηχανήματα που θα χρησιμοποιηθούν ως workers όπως και στο μηχάνημα του master. Ξεκινάμε με την εγκατάσταση της Java.

1. Εγκατάσταση Java.

   i) Έλεγχος αν υπάρχει εγκατεστημένη η Java: `java -version`

   ii) Εάν δεν επιστραφεί απάντηση όπως η παρακάτω, συνεχίζουμε στο (iii):

```
openjdk version "1.8.0_131"
OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)
OpenJDK Server VM (build 25.131-b11, mixed mode)
```

   iii) Εγκαθιστούμε την Java με την εντολή: `sudo apt install default-jre`

2. Εγκατάσταση Scala.

i) Έλεγχος αν υπάρχει εγκατεστημένη η Scala: `scala -version`

ii) Εάν δεν επιστραφεί απάντηση όπως η παρακάτω, συνεχίζουμε στο (iii):

```
Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
```

iii) Κατεβάζουμε το πακέτο της Scala από εδώ.

iv) Εκτελούμε:

```
sudo mv scala-2.11.6.tgz /usr/local/
cd /usr/local/
sudo tar xvfz scala-2.11.6.tgz
sudo nano ~/.bashrc
```

και προσθέτουμε στο τέλος του αρχείου τις γραμμές:

```
export SCALA_HOME=/usr/local/scala-2.11.6
export PATH=$PATH:$SCALA_HOME/bin
```

εκτελούμε: `source ~/.bashrc`

```
scala -version
```

3. Εγκατάσταση Spark.

i) Κατεβάζουμε το πακέτο του Spark από εδώ.

ii) Εκτελούμε:

```
sudo mv spark-2.1.0-bin-hadoop2.7.tgz /usr/local/
cd /usr/local/
sudo tar xvfz spark-2.1.0-bin-hadoop2.7.tgz
sudo mv spark-2.1.0-bin-hadoop2.7 spark2.1.0
sudo nano ~/.bashrc
```

και προσθέτουμε στο τέλος του αρχείου την γραμμή:

```
export PATH=$PATH:/usr/local/spark2.1.0/bin
```

εκτελούμε: `source ~/.bashrc`

```
spark-shell
```

iii) Μετά την εκκίνηση του `spark-shell` βλέπουμε:

```
               :~$ spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/10/04 15:12:58 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where a
pplicable
17/10/04 15:12:59 WARN Utils: Your hostname, pcnikos resolves to a loopback address: 127.0.1.1; using 192.168.1.26 instead (on interf
ace enp1s5)
17/10/04 15:12:59 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
17/10/04 15:13:37 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so
recording the schema version 1.2.0
17/10/04 15:13:38 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
17/10/04 15:13:42 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://192.168.1.26:4040
Spark context available as 'sc' (master = local[*], app id = local-1507119182861).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (OpenJDK Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Configuration files του Spark:

1. Για τη εύκολη επικοινωνία των μηχανημάτων, τροποποιούμε το αρχείο σε κάθενα από αυτά ως εξής: Βάζουμε σχόλιο στην IP "127.0.1.1", συνήθως βρίσκεται στη 2η σειρά, και επεκτείνουμε το αρχείο κάτω από αυτήν με τις γραμμές:

```
147.102.19.151	master	kapsoulis1-VirtualBox
147.102.19.152	slave1	kapsoulis2-VirtualBox
147.102.19.199	slave2	kapsoulis3-VirtualBox
```

Κάθε συμβολοσειρά πρέπει να διαχωρίζεται από την άλλη με "Tab".

2. Για τον έλεγχο των workers, μορφοποιούμε το αρχείο του master: "/usr/local/spark/conf/slaves".
Σε κάθε IP που βρίσκεται σε αυτό το αρχείο, ο master θα σηκώσει έναν slave/worker. Αν αυτό δεν θέλουμε να συμβεί, βάζουμε την IP σε σχόλιο με prefix τον χαρακτήρα '#'.

Εγκατάσταση MongoDB και Sharded Cluster:

Η εγκατάσταση της MongoDB είναι αρκετά απλή, εύκολη και γρήγορη.

1. Εγκατάσταση της βάσης. Εκτελούμε τις εντολές:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927udo
echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
sudo nano /etc/systemd/system/mongodb.service
```

βάζοντας στο αρχείο “mongodb.service” τις παρακάτω σειρές:

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
[Service]
```

```
User=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf
[Install]
WantedBy=multi-user.target
```

συνεχίζουμε με τις εντολές εκκίνησης και ελέγχου λειτουργίας:

```
sudo systemctl start mongodb
sudo systemctl status mongodb
sudo systemctl enable mongodb
```

2. Sharding Configuration. Σε όλα τα μηχανήματα που θα δουλεύψουν ως shards πάνω στο Sharded Cluster, εκτελούμε τα παρακάτω:

```
sudo su
mkdir -p /var/log/mongodb_sharded/
chown mongodb:mongodb /var/log/mongodb_sharded/
mkdir -p /var/lib/mongodb_sharded/shardsrv
chown mongodb:mongodb /var/lib/mongodb_sharded/shardsrv
openssl rand -base64 756 > ~/mongo.keyfile
chmod 400 ~/mongo.keyfile
nano /etc/mongod_shardsrv.conf
```

>>>>> Και κάνουμε επικόλληση το παρακάτω:

```
# shardsrv.conf
# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/
sharding:
  clusterRole: shardsvr
# Where and how to store data.
storage:
  dbPath: /var/lib/mongodb_sharded/shardsrv
  journal:
    enabled: true
#  engine:
#  mmapv1:
#  wiredTiger:
# where to write logging data.
systemLog:
  destination: file
```

```
    logAppend: true
    path: /var/log/mongodb_sharded/mongod_shard.log
  # network interfaces
  net:
    port: 27020                        #  Η θύρα που ακούει το shard
    bindIp: 127.0.0.1,147.102.19.151   #  Η IP address του shard
  #processManagement:
  security:
   keyFile: /home/kapsoulis/mongo.keyfile
   #  όπου /home/kapsoulis/mongo.keyfile = ~/mongo.keyfile
  #operationProfiling:
  #replication:
  #sharding:
  ## Enterprise-Only Options:
  #auditLog:
  #snmp:
```

<<<<< Τέλος επικόλλησης. Ανοίγουμε ένα terminal και τρέχουμε:

```
sudo mongod --config /etc/mongod_shardsrv.conf
```

3. Επιπλέον για το μηχάνημα που θα τρέχει και configuration server και query router, συνεχίζουμε με τα παρακάτω:

```
sudo su
mkdir -p /var/lib/mongodb_sharded/configsrv
chown mongodb:mongodb /var/lib/mongodb_sharded/configsrv
nano /etc/mongod_confidsrv.conf
```

>>>>> Και κάνουμε επικόλληση το παρακάτω:

```
  # configsrv.conf
  # for documentation of all options, see:
  #   http://docs.mongodb.org/manual/reference/configuration-options/
  sharding:
     clusterRole: configsvr
  # Where and how to store data.
  storage:
    dbPath: /var/lib/mongodb_sharded/configsrv
    journal:
      enabled: true
```

```
#  engine:
#  mmapv1:
#  wiredTiger:
# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb_sharded/mongod_config.log
# network interfaces
net:
  port: 27019                                          #  Θύρα confid server
  BindIp: 127.0.0.1,147.102.19.151                     #  IP address config server
#processManagement:
security:
  keyFile: /home/kapsoulis/mongo.keyfile               #  Τοποθεσία mongo.keyfile
#operationProfiling:
#replication:
#sharding:
## Enterprise-Only Options:
#auditLog:
#snmp:
```

<<<<< Τέλος επικόλλησης.

```
nano /etc/mongos.conf
```

>>>>> Και κάνουμε επικόλληση το παρακάτω:

```
# mongos.conf
# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/
sharding:
  configDB: 147.102.19.151:27019          #  Διεύθυνση και θύρα config server
# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb_sharded/mongos.log
# network interfaces
net:
```

```
  port: 27021
  bindIp: 127.0.0.1,147.102.19.151
#processManagement:
security:
  keyFile: /home/kapsoulis/mongo.keyfile # Τοποθεσία mongo.keyfile
#operationProfiling:
#replication:
#sharding:
## Enterprise-Only Options:
#auditLog:
#snmp:
```

<<<<< Τέλος επικόλλησης.

Ανοίγουμε ένα νέο terminal και τρέχουμε:

```
sudo mongod --config /etc/mongod_configsrv.conf
```

Ανοίγουμε ένα ακόμη terminal και τρέχουμε:

```
sudo mongod --config /etc/mongos.conf
```

Εισαγωγή Προϊόντων στη βάση:

Προτού εισάγουμε τα προϊόντα, πρέπει να συνδεθούμε στη διαχειριστική βάση με την εντολή:

```
mongo --port 27021 -u vrettos -p vrettos --authenticationDatabase admin
```

να δημιουργήσουμε `username` και `password` για τη βάση `recommender`:

```
db.createUser({user:"vrettos",pwd:"vrettos", roles: [{role:"readWrite",db:"recommender"}]})
```

και έπειτα να εκτελέσουμε:

```
sh.shardCollection("recommender.products",{price:1})
```

και μετά να εισάγουμε το αρχείο των δεδομένων μας:

```
mongoimport -h 147.102.19.151 --port 27021 --username vrettos --password vrettos --db recommender --collection products --jsonArray products.json
```

Έναρξη Apache Spark:

Στο μηχάνημα του master εκτελούμε:

`nano /usr/local/spark/conf/slaves` # Επιλέξτε πόσοι workers θα σηκωθούν, για όσους δεν θα σηκωθούν, βάλτε την IP τους σε σχόλιο με τον χαρακτήρα '#'.

```
/usr/local/spark/sbin/start-all.sh
```

Υποβολή εφαρμογής στο Spark:

Από το μηχάνημα του master εκτελούμε:

```
spark-submit --class "ProductRecommender"  ProductRecommenderSystem.jar
```

Η παρακάτω εικόνα είναι η επόμενη οθόνη που θα δει ο πελάτης:

```
kapsoulis@kapsoulis1-VirtualBox:~$ spark-submit --class "ProductRecommender"  ProductRecommenderSystem.jar

* * * What would you like your service to have? * * *
encryption, compression
```

Εδώ ο πελάτης πληκτρολόγησε "encryption, compression", και μόλις πατήσει "Enter" θα ξεκινήσει η εφαρμογή την εκτέλεσή της.

Βιβλιογραφία – Αναφορές

[1] *National Institute of Standards and Technology, Information Technology Laboratory*: The NIST Definition of Cloud Computing, September 2011

[2] Security Guidance for Critical Areas of Focus in Cloud Computing V3.0

[3] Cisco Blogs: Pentesting in the Cloud

http://www.haikumind.com/cloud-computing-acronyms-iaas-paas-and-saas/

[4] Metadata

[5] NISO Metadata

[6] BTRFS

[7] Kimball DW/BI System Architecture

[8] OASIS

[9] TOSCA Simple Profile in YAML version 1.1

[10] Παράδειγμα από το official documentation του [9]

[11] YAML vs JSON

[12] YAML Wikipedia

[13] JSON Official Site

[14] YAML relation to JSON

[15] A Relational Model of Data for Large Shared Data Banks

[16] Relational Database Constraints

[17] SQL

[18] Oracle Data Mining blog

[19] Companies and Organizations using Apache Spark

[20] Resilient Distributed Datasets: A Tolerant Fault Abstraction for Computing Cluster In Memory

[21] Apache Spark's GraphX

[22] Apache Spark: Cluster Mode Overview

[23] Generate random number between given range.

[24] Recommendation Systems Stanford – Content-based Filtering: 9.2.4 Representing Item Profiles

[25] Stanford Cosine Similarity

[26] Cosine Similarity Wikipedia

[27] Apache Spark: SparkSession

[28] MongoDB Aggregation Pipeline

[29] Big Data Wikipedia Definition

[30] Big Data SAS Company Definition

[31] Big Data 3 Vs

[32] Εκτίμηση ετήσιου ρυθμού παραγωγής δεδομένων το 2025

[33] IBM Research: 10 Key Marketing Trends for 2017

[34] Hours of video uploaded to YouTube every minute

[35] Global mobile data traffic from 2016 to 2021

[36] IoT: Major Cause of Big Data Increase

[37] Big Data Velocity: Google Received Queries

[38] Facebook Scaling DW Up To 300 Petabytes