ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτόματη Παροχή Πόρων στο Περιβάλλον Κατανεμημένης Επεξεργασίας Storm

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Ν. Γκολέμης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Αθήνα, Απρίλιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αυτόματη Παροχή Πόρων στο Περιβάλλον Κατανεμημένης Επεξεργασίας Storm

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Ν. Γκολέμης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Απριλίου 2017.

....	....	....
Ν. Κοζύρης	Γ. Γκούμας	Δ. Τσουμάκος
Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ι.Π

Αθήνα, Απρίλιος 2017

....

Ευάγγελος Ν. Γκολέμης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Copyright © Ευάγγελος Ν. Γκολέμης, 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.

Περίληψη

Στη σύγχρονη εποχή παράγονται συνεχώς δεδομένα σε μεγάλο όγκο και οι ρυθμοί παραγωγής αυξάνονται καθημερινά. Είναι λογικό, λοιπόν, να δημιουργείται η ανάγκη της άμεσης εξαγωγής χρήσιμης πληροφορίας και συμπερασμάτων από το σύνολο των παραγόμενων δεδομένων.
Η επεξεργασία ροών δεδομένων είναι ένα μοντέλο επεξεργασίας που καλύπτει την παραπάνω ανάγκη. Τα δεδομένα μοντελοποιούνται ως συνεχόμενες ροές και η επεξεργασία τους διαιρείται σε επεξεργαστικά στάδια. Έχουν δημιουργηθεί αρκετά συστήματα επεξεργασίας ροών δεδομένων. Στην παρούσα διπλωματική θα χρησιμοποιήσουμε το Apache Storm.
Ανεξάρτητα από το σύστημα που χρησιμοποιείται, η βελτιστοποίηση των επιδόσεων των εφαρμογών είναι ο πιο σημαντικός στόχος. Αυτός θα είναι και ο στόχος της διπλωματικής, η ανάπτυξη συστήματος που θα βελτιστοποιεί τις επιδόσεις μιας εφαρμογής στο σύστημα κατανεμημένης επεξεργασίας ροών δεδομένων Storm. Πιο συγκεκριμένα, θα δέχεται μια εφαρμογή και θα παρέχει τις κατάλληλες παραμέτρους παραλληλίας.
Στα πλαίσια της διπλωματικής θα αναπτύξουμε δύο μηχανισμούς που θα έχουν τον ίδιο στόχο όπως τον παρουσιάσαμε παραπάνω, αλλά θα είναι διαφορετικοί δομικά και θα προσφέρουν διαφορετικού είδους πλεονεκτήματα ο καθένας. Ο πρώτος θα είναι ένας "dynamic" μηχανισμός, θα τρέχει παράλληλα(online) με την εφαρμογή για να βελτιστοποιήσει τις επιδόσεις της και θα παρέχει elasticity. Ο δεύτερος θα είναι ένας "static" μηχανισμός που θα περνάει από ένα machine learning στάδιο και θα μπορεί να παρέχει άμεσα και σε ένα βήμα τη βέλτιστη λύση.

Λέξεις Κλειδιά
Επεξεργασία Ροών Δεδομένων, Παράλληλα Συστήματα, Κατανεμημένα Συστήματα, Υπολογιστικό Νέφος, Apache Storm, Μαζικά Δεδομένα, Μηχανική Μάθηση, Ελαστικότητα.

Abstract

In the modern world, data is continually produced in large volumes and production rates are increasing daily. It is therefore reasonable to create the need to directly extract useful information and conclusions from all the data produced.
Stream processing is a processing model that addresses the above need. The data is calculated as continuous streams and their processing is divided into processing steps. Several stream processing systems have been created. Apache Storm is the system that is going to be used in this diploma thesis.
Regardless of the system used, optimization of application performance is of the most importance and it is going to be the goal of the diploma thesis. We will develop a system that will optimize the performance of an application in the distributed stream processing system Storm. In particular, the system introduced will take as input an application and provide the appropriate parallelization parameters.
Within the framework of this diploma thesis, two mechanisms will be introduced and developed that will have the same goal as presented above but each one will be structurally different and will offer different kinds of advantages. The first will be a "dynamic" mechanism, running on-line with the application to optimize its performance and provide elasticity. The second will be a "static" mechanism that will go through a machine learning stage and will be able to offer the best solution at once.

Keywords
Stream Processing, Distributed Systems, Parallel Systems, Cloud, Apache Storm, Big Data, Machine Learning, Elasticity.

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον Καθηγητή Ε.Μ.Π Νεκτάριο Κοζύρη, ο οποίος ήταν ο επιβλέποντας της διπλωματικής μου εργασίας και μου έδωσε την ευκαιρία να συνεργαστώ με το εργαστήριο υπολογιστικών συστημάτων. Καθ' όλη τη διάρκεια των σπουδών μου τα μαθήματα του κ. Κοζύρη αποτέλεσαν σημείο αναφοράς και έμπνευσης, εξερευνώντας σύγχρονα θέματα της επιστήμης των υπολογιστών.
Επίσης, θα ήθελα να ευχαριστήσω όλους τους συναδέλφους και τα μέλη του εργαστηρίου υπολογιστικών συστημάτων για τη συνεργασία και ιδιαίτερα την Κατερίνα Δόκα της οποίας η καθοδήγηση και οι συμβουλές ήταν ιδιαιτέρως χρήσιμες κατά τη διάρκεια της εκπόνησης της διπλωματικής εργασίας.

Περιεχόμενα

1 Εισαγωγή ... 1
1.1 Σύγχρονα Δεδομένα (Big Data Era) ... 1
1.2 Stream Computing (Επεξεργασία ροών) ... 2
1.2.1 Βασικές έννοιες ... 2
1.2.2 Αποτελεσματικότητα και πλεονεκτήματα ... 3
1.2.3 Μοντελοποίηση περιβάλλοντος και εφαρμογών ... 4
1.3 Σχετικά με την εργασία ... 8
1.3.1 Περιβάλλον έρευνας ... 8
1.3.2 Στόχος της διπλωματικής ... 9
1.3.3 Δομή διπλωματικής ... 10
2 Ανάλυση Περιβάλλοντος ... 10
2.1 Βασικές έννοιες ... 10
2.2 Αρχιτεκτονική του Apache Storm ... 13
2.2.1 Σχεδιαστικές επιλογές ... 13
2.2.2 Κατανόηση των εσωτερικών λειτουργιών ... 15
2.3 Ορισμός προβλήματος ... 28
3 Αρχιτεκτονική Συστήματος ... 29
3.1 Monitoring ... 29
3.2 Automatic Scaling (Rule Based) ... 34
3.2.1 Στόχος ... 34
3.2.2 Βασική λογική ... 34
3.3 Αρχιτεκτονική ... 36
3.4 Configuration Finder (Machine Learning) ... 41
3.4.1 Στόχος ... 41
3.4.2 Βασική λογική ... 41
3.4.3 Αρχιτεκτονική ... 43
4 Πειραματική Αποτίμηση ... 53
4.1 Πειραματικό Περιβάλλον ... 54
4.2 Ανάλυση των Αποτελεσμάτων ... 56
4.2.1 WordCount Topology ... 57
4.2.2 CyberShake Topology ... 62
4.2.3 Montage Topology ... 67
4.2.4 Matrix Topology ... 72

4.3 Συμπεράσματα 76
5 Βιβλιογραφία..78

1 Εισαγωγή

1.1 Σύγχρονα Δεδομένα (Big Data Era)

Ζούμε σε έναν κόσμο που όλο και περισσότερο συνδέεται και ενορχηστρώνεται μέσω της πληροφορίας, η έννοια της οποίας είναι αλληλένδετη με αυτή των δεδομένων. Είναι γεγονός ότι η εποχή μας είναι αυτή των “Big Data”, δηλαδή δεδομένων μεγάλων σε όγκο και πολυπλοκότητα -είτε δομημένων είτε αδόμητων- που παράγονται και είναι διαθέσιμα σε καθημερινή βάση από διάφορες πηγές. Γεγονός επίσης είναι ότι ο όγκος των διαθέσιμων δεδομένων-πληροφορίας καθώς και ο ρυθμός παραγωγής αυτών αυξάνεται συνεχώς.

Το σημαντικό, βέβαια, δεν είναι ο όγκος των διαθέσιμων δεδομένων, αλλά ο τρόπος που τα διαχειρίζεται κανείς για να βγάλει χρήσιμα συμπεράσματα ή γενικότερα να επωφεληθεί. Είναι λογικό, λοιπόν, η προσοχή της πληροφορικής να στραφεί προς την επεξεργασία τέτοιου είδους δεδομένων. Η σημαντικότητα της επεξεργασίας αυτών των δεδομένων φαίνεται από τις πηγές που προέρχονται καθώς και το εύρος τους. Ενδεικτικά, τα δεδομένα που συγκεντρώνονται μπορούν να προέρχονται από κοινωνικά δίκτυα, χρηματοοικονομικές αγορές, τηλεπικοινωνίες, διάφορα ήδη αισθητήρων, ακόμα και από machine-to-machine επικοινωνίες. Ο συνδυασμός πληροφοριών από ένα τόσο μεγάλο εύρος πηγών έχει πολύ μεγάλες προοπτικές και μπορεί να προσφέρει πολύ σημαντικά ευρήματα.

Γενικά, υπάρχουν δύο μηχανισμοί για big data computing και αυτοί είναι big data stream computing και big data batch computing. Το πρώτο είναι ένα μοντέλο απευθείας και συνεχόμενης επεξεργασία των δεδομένων, ενώ το δεύτερο είναι το γνωστό μας μοντέλο αποθήκευσης των δεδομένων και εν συνεχεία επεξεργασίας (store – compute), όπως για παράδειγμα το MapReduce.
Σε πολλές περιπτώσεις το δεύτερο μοντέλο επεξεργασίας δεν επαρκεί, ειδικά όταν πρόκειται για εφαρμογές real-time, όπου τα δεδομένα μεταβάλλονται συχνά κατά την πάροδο του χρόνου και η πιο πρόσφατη πληροφορία είναι και η σημαντικότερη, π.χ. financial trades, sensor data tracking κτλ. Τέτοιες εφαρμογές μπορούν να εξυπηρετηθούν καλύτερα από το πρώτο μοντέλο επεξεργασίας (big data stream computing) γιατί τα δεδομένα έχουν τη μορφή ροών.

1.2 Stream Computing (Επεξεργασία ροών)

1.2.1 Βασικές έννοιες

Η επεξεργασία ροών είναι ένα μοντέλο αποτελεσματικής επεξεργασίας μεγάλου όγκου δεδομένων προσφέροντας μηχανισμούς επεξεργασίας χαμηλής καθυστέρησης (low-latency) και μαζικής παραλληλίας.

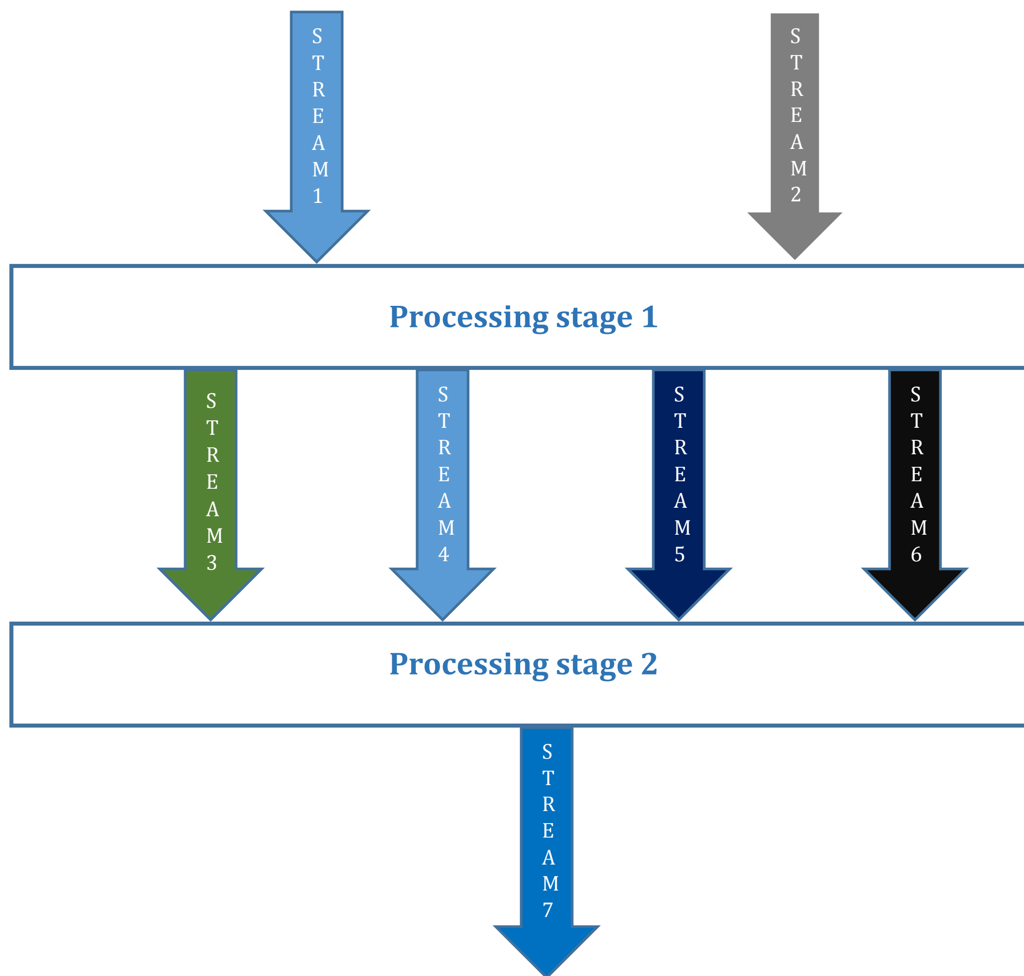
Η πληροφορία εισάγεται και επεξεργάζεται σε μορφή συνεχόμενων και θεωρητικά άπειρων ροών δεδομένων (data streams).
Κάθε ροή έχει τα εξής χαρακτηριστικά:

- **Ρυθμό**: η ταχύτητα με την οποία έρχονται τα δεδομένα
- **Μέγεθος**: ο όγκος των δεδομένων είναι θεωρητικά άπειρος
- **Περιεχόμενο**: το είδος των δεδομένων που περιέχει η ροή
- **Πραγματικός χρόνος**: τα δεδομένα της ροής παράγονται και πρέπει να επεξεργαστούν σε πραγματικό χρόνο, άρα χρειάζεται χαμηλή καθυστέρηση

Η επεξεργασία των ροών δεν είναι απαραίτητο να γίνει σε ένα στάδιο. Το αντίθετο μάλιστα, είναι επιθυμητό να υπάρχουν πολλαπλά στάδια επεξεργασίας χαμηλής καθυστέρησης.
Αυτά τα στάδια είναι διατεταγμένα μεταξύ τους, με την έννοια ότι οι ροές κατευθύνονται από τα λογικά ανώτερα στάδια προς τα λογικά κατώτερα στάδια. Κάθε ανώτερο στάδιο επεξεργάζεται κατάλληλα την πληροφορία και παράγει μία η περισσότερες νέες ροές δεδομένων που κατευθύνονται προς τα κατώτερα στάδια επεξεργασίας.

Οι ροές που εισέρχονται και εξέρχονται δεν είναι απαραίτητο να έχουν τα ίδια χαρακτηριστικά.

Ένα παράδειγμα της δομής σε επεξεργαστικά στάδια και των ροών μεταξύ τους φαίνεται στο σχήμα 1.1 παρακάτω.



Σχήμα 1.1 – Processing stages and streams

1.2.2 Αποτελεσματικότητα και πλεονεκτήματα

Τα "Big Data" είναι ένα σύνολο δεδομένων (data set) που χαρακτηρίζεται από μεγάλο ρυθμό, μεγάλο όγκο, τις περισσότερες φορές είναι αδόμητα και προέρχονται από πολλαπλές πηγές σε πραγματικό χρόνο. Επίσης, λόγω του μεγάλου όγκου τους είτε δεν είναι δυνατό να αποθηκευτούν στο σύνολό τους είτε δεν είναι αναγκαίο να αποθηκευτεί τόσο μεγάλη πληροφορία αλλά προτιμάται να φιλτραριστεί σε πραγματικό χρόνο. [1]

Εύκολα μπορεί να παρατηρηθεί ότι τα big data ταιριάζουν με τις ροές δεδομένων τόσο σε μορφή όσο και σε συμπεριφορά.
Σχεδόν το σύνολο της πληροφορία που παράγεται στις μέρες μας έχει τη μορφή των συνεχόμενης ροής δεδομένων κατευθείαν από την πηγή της ή μπορεί να έρθει σε μορφή ροής πολύ εύκολα.

Σε αυτό το σημείο είναι εμφανές ότι το μοντέλο της επεξεργασίας ροών εξυπηρετεί με αποτελεσματικό τρόπο τα δεδομένα με τη μορφή που παράγονται σήμερα και κατά συνέπεια βρίσκει όλο και περισσότερα πεδία εφαρμογών.

Συγκεκριμένα, τα πλεονεκτήματα του μοντέλου επεξεργασίας ροών δεδομένων είναι:

- Δυνατότητα επεξεργασίας πολλαπλών ροών με διαφορετικά χαρακτηριστικά ταυτόχρονα
- Επεξεργασία των δεδομένων κατευθείαν μετά τη δημιουργία τους
- Άμεσα αποτελέσματα-συμπεράσματα
- Εύκολο και γρήγορο φιλτράρισμα της επιπλέον πληροφορίας
- Διαμοιρασμός της λογικής σε τμήματα μικρής καθυστέρησης

Είναι λογικό, λοιπόν, η επεξεργασία ροών μεγάλων δεδομένων (big data stream computing) είναι η σύγχρονη τάση για πολλούς τομείς που χρειάζεται να βγάλουν συμπεράσματα σε πραγματικό χρόνο. Για παράδειγμα:

- Οικονομικές αγορές: ανάλυση ρίσκου, marketing, business intelligence, fraud detection(pattern recognition)
- Internet: κοινωνικά δίκτυα, μηχανές αναζήτησης, network monitoring
- Internet of Things: environmental monitoring, smart cities

1.2.3 Μοντελοποίηση περιβάλλοντος και εφαρμογών

1.2.3.1 Μοντελοποίηση προβλημάτων-εφαρμογών

Η μοντελοποίηση των εφαρμογών-προβλημάτων στην επεξεργασία ροών μπορεί να γίνει μέσω κατευθυνόμενων απεριοδικών γράφων (directed acyclic graphs). Κάθε τέτοιος γράφος περιλαμβάνει ένα σύνολο από κορυφές(vertices) παριστάνει μια επεξεργαστική μονάδα και ένα σύνολο από κατευθυνόμενες ακμές(edges) που παριστάνουν τις ροές δεδομένων.

$$G = \big(V(G), E(G)\big), \qquad V(G) = \{v_1, v_2, \dots, v_n\}, E(G) = \big\{e_{1,2}, e_{1,3}, \dots, e_{n-1,n}\big\}$$

$$\exists e_{i,j} \in E(G) \Rightarrow v_i, v_j \in V(G), v_i \neq v_j \Rightarrow \langle v_i \xrightarrow{data} v_j \rangle$$

Παραπάνω, φαίνεται ο ορισμός του directed acyclic graph(DAG).

Ο βαθμός εισόδου(input degree) μιας κορυφής v_i είναι ο αριθμός των εισερχόμενων ακμών.
Ο βαθμός εξόδου (output degree) μιας κορυφής v_i είναι ο αριθμός των εξερχόμενων ακμών.
Μια κορυφή v_i ονομάζεται πηγή(source) αν ο βαθμός εισόδου είναι 0.
Μια κορυφή v_i ονομάζεται απόληξη(end) αν ο βαθμός εξόδου είναι 0.

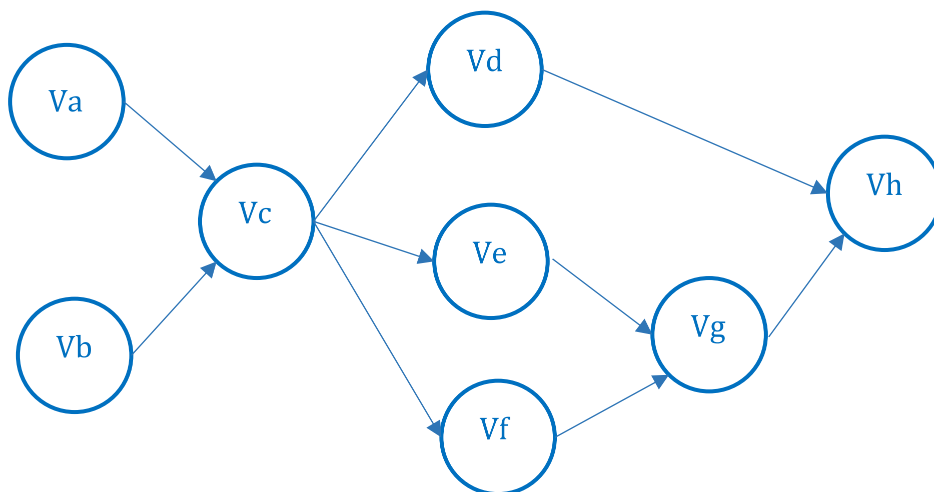
Ένα κατευθυνόμενο μονοπάτι (directed path) είναι ένα σύνολο από ακμές που ενώνουν μια σειρά από κορυφές. Επειδή μάλιστα είναι DAG ικανοποιείται αυτόματα ο περιορισμός για επανάληψη ίδιων κορυφών, αφού δεν υπάρχουν

κύκλοι στον γράφο. Μεταξύ 2 κορυφών v_i , v_j μπορούν να υπάρχουν περισσότερα του ενός μονοπάτια.
Η καθυστέρηση(latency) ενός μονοπατιού $p(v_i, v_j)$ είναι το άθροισμα όλων των καθυστερήσεων από τις κορυφές c_{v_i} και τις ακμές $c_{e_{i,j}}$ και δίνεται από την εξίσωση:

$$l_p(v_i, v_j) = \sum_{v_x \epsilon V(p(v_i,v_j))} c_{v_x} + \sum_{e_{x,y} \epsilon V(p(v_i,v_j))} c_{e_{x,y}}$$

Το κρίσιμο μονοπάτι(critical path) ενός γράφου G είναι το μονοπάτι με τη μέγιστη καθυστέρηση με αρχή κάποια πηγή v_s και τέλος κάποια απόληξη v_e. Αν υπάρχουν k μονοπάτια $p(v_s, v_e)$ τότε η καθυστέρηση του κρίσιμου μονοπατιού είναι και η καθυστέρηση του γράφου ορίζεται ως:

$$l(G) = \max\{l_{p_1}(v_s, v_e), l_{p_2}(v_s, v_e), \dots, l_k(v_s, v_e)\}$$



Σχήμα 1.2 – DAG example

Στο σχήμα 1.2 φαίνεται ένα παράδειγμα κατευθυνόμενου απεριοδικού γράφου 8 κορυφών. Πάνω σε αυτόν μπορούμε να σημειώσουμε τα εξής:

- Το σύνολο των κορυφών είναι $V(G) = \{v_a, v_b, \dots, v_h\}$
- Το σύνολο των κατευθυνόμενων ακμών είναι $E(G) = \{e_{ac,2}, e_{b,c}, \dots, e_{g,h}\}$
- Για την κορυφή v_c βαθμός εισόδου = 2 και βαθμός εξόδου = 3
- Οι κορυφές v_a, v_b είναι πηγές
- Η κορυφή v_h είναι απόληξη
- Μεταξύ των κορυφών v_c, v_h υπάρχουν 3 μονοπάτια.

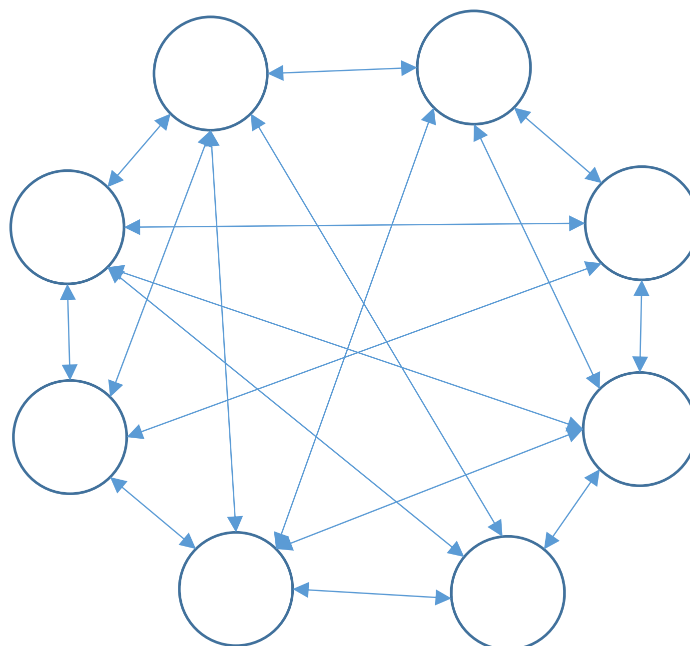
1.2.3.2 Μοντελοποίηση περιβάλλοντος

Το μοντέλο επεξεργασίας ροών διαχειρίζεται μεγάλο όγκο δεδομένων σε μορφή ροών τα οποία επεξεργάζονται σε πραγματικό χρόνο και τα αποτελέσματα θα πρέπει να είναι επίσης σε πραγματικό χρόνο.

Στις ροές δεδομένων, η πληροφορία έρχεται με μεγάλους ρυθμούς, δεν έχει συγκεκριμένη δομή, είναι ευμετάβλητη και φυσικά είναι θεωρητικά άπειρη. Κατά συνέπεια, κάθε περιβάλλον επεξεργασίας ροών δεδομένων θα πρέπει να μοντελοποιηθεί και να σχεδιαστεί λεπτομερώς. Ανάλογα με τους στόχους που έχουν τεθεί για το τελικό σύστημα και τις ανάγκες που θέλει να εξυπηρετήσει πρέπει να βελτιστοποιηθούν όλοι οι απαραίτητοι τομείς του περιβάλλοντος. Ορισμένοι βασικοί τομείς είναι αυτοί της δομής του συστήματος(system structure), της μεταφοράς δεδομένων(data transmission) και της διαθεσιμότητας(high availability).

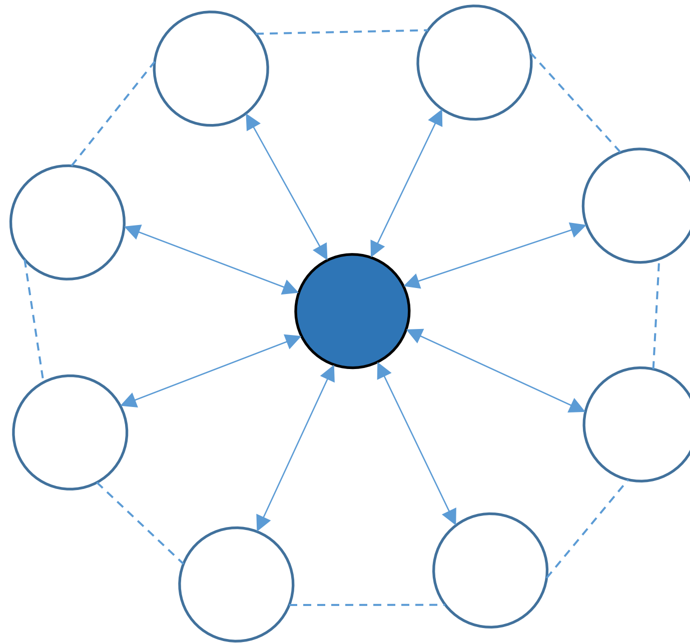
Δομή Συστήματος

Οι δύο βασικές δομές είναι η συμμετρική και η master-slave και φαίνονται στα σχήματα 1.3 και 1.4 αντίστοιχα.



Σχήμα 1.3 – Symmetric structure

Στη συμμετρική δομή όλοι οι κόμβοι είναι ισότιμοι. Κατά συνέπεια, είναι εύκολο να προστεθούν ή να αφαιρεθούν κόμβοι για να βελτιωθεί η κλιμάκωση του συστήματος. Όμως, ορισμένες συστημικές λειτουργίες που περιλαμβάνουν το σύνολο των κόμβων όπως δέσμευση πόρων, ανοχή σε σφάλματα, εξισορρόπηση φόρτου είναι δύσκολο χωρίς την ύπαρξη κεντρικού κόμβου.



Σχήμα 1.4 – Master - slave structure

Στη δομή master-slave ένας από τους κόμβους έχει τη δουλειά του διαχειριστή και οι υπόλοιποι είναι οι εργάτες. Ο διαχειριστής είναι υπεύθυνος για τον έλεγχο του συστήματος(δέσμευση πόρων, ανοχή σε σφάλματα, εξισορρόπηση φόρτου). Κάθε κόμβος-εργάτης εκτελεί κάποιες συγκεκριμένες εργασίες που του ανατίθενται από τον διαχειριστή. Η ροή των δεδομένων γίνεται από τους εργάτες προς το διαχειριστή και αντίστροφα. Έτσι ο κόμβος-διαχειριστής είναι bottleneck καθώς και single point of failure. [1]

<u>Μεταφορά Δεδομένων</u>

Η μεταφορά των δεδομένων γίνεται από τους κόμβους που ανήκουν σε ανώτερο επεξεργαστικό στάδιο προς τους κόμβους που ανήκουν σε κατώτερο επεξεργαστικό στάδιο. Υπάρχουν 2 διαφορετικοί τρόποι για την μεταφορά δεδομένων μέσω ροών:

- Push system: Κάθε φορά που τελειώνει η επεξεργασία δεδομένων σε κόμβο ανώτερου επιπέδου, γίνεται αμέσως προώθηση στους κόμβους του κατώτερου επιπέδου, Αυτό θέλει προσοχή γιατί σε περίπτωση που ο κόμβος-προορισμός των δεδομένων μπορεί να είναι απασχολημένος ή να έχει αποτύχει με αποτέλεσμα να χαθούν δεδομένα
- Pull system: Ο κόμβος του κατώτερου επιπέδου ζητάει δεδομένα από τον αντίστοιχο ανώτερου επιπέδου. Αν δεν ζητηθούν δεδομένα από κατώτερο επίπεδο, ο κόμβος του ανώτερου επιπέδου αποθηκεύει προσωρινά τα δεδομένα μέχρι αυτά να ζητηθούν. Υπάρχει κίνδυνος δεδομένα να περιμένουν αποθηκευμένα μεγάλο χρονικό διάστημα και να λήξουν.

Διαθεσιμότητα
Η δημιουργία αντιγράφων ασφαλείας είναι ο κύριος τρόπος για να υπάρχει συνεχής διαθεσιμότητα του συστήματος. Υπάρχουν 3 στρατηγικές για να παρέχεται συνεχής διαθεσιμότητα:

- Παθητική αναμονή: Κάθε κύριος κόμβος στέλνει checkpoint data σε έναν backup κόμβο. Αν ο κύριο κόμβος αποτύχει, ο backup κόμβος αναλαμβάνει από το τελευταίο checkpoint.
- Ενεργητική Αναμονή: Οι δευτερεύοντες-backup κόμβοι επεξεργάζονται το σύνολο των δεδομένων που επεξεργάζεται και οι κύριοι. Αυτό προσφέρει πολύ γρήγορο recovery time αλλά δεσμεύει πολλούς πόρους.
- Upstream backup: Οι κόμβοι των ανώτερων επιπέδων χρησιμοποιούνται και σαν αντίγραφα ασφαλείας για τους κόμβους των κατώτερων επιπέδων. Αν κάποιος κόμβος αποτύχει, ο αντίστοιχος κόμβος του ανώτερου επιπέδου επαναλαμβάνει τα δεδομένα. Αυτός ο τρόπος είναι ο πιο ελαφρύς στο runtime, αλλά έχει μεγάλο recovery time.

1.3 Σχετικά με την εργασία

1.3.1 Περιβάλλον έρευνας

Το σύστημα επεξεργασίας ροών δεδομένων που θα χρησιμοποιηθεί είναι το Apache Storm. Επιλέχθηκε έναντι των υπόλοιπων συστημάτων γιατί είναι ίσως το πιο καθαρόαιμο, εκτελώντας την επεξεργασία των ροών δεδομένων ανά μοναδιαία είσοδο.

Το Apache Storm είναι ένα κατανεμημένο περιβάλλον επεξεργασίας ροών δεδομένων. Ως σύστημα παρέχει εγγυήσεις για αξιόπιστη επεξεργασία των δεδομένων καθώς και ανοχή σε σφάλματα. Η δημιουργία και εκτέλεση εφαρμογών απαιτεί βασική κατανόηση της λογικής επεξεργασίας ροών. Κανείς μπορεί να ξεκινήσει από τη δημιουργία απλών εφαρμογών και να φτάσει μέχρι ιδιαίτερα πολύπλοκες λύσεις που συνυπάρχουν και ενώνουν πολλαπλά εργαλεία που βασίζονται στη λογική των κατανεμημένων συστημάτων και της επεξεργασίας ροών δεδομένων. Τέλος, παρέχει ένα φιλικό προς τον χρήστη περιβάλλον παρακολούθησης των εφαρμογών του καθώς τρέχουν στο σύστημα.

Το σύνολο της διπλωματικής θα στηριχθεί στο παραπάνω σύστημα. Σε επόμενο κεφάλαιο θα αναλύσουμε τα σημαντικά κομμάτια και τις ουσιαστικότερες ιδέες που αποτελούν τη βάση του συστήματος.

1.3.2 Στόχος της διπλωματικής

1.3.2.1 Περιγραφή της ανάγκης

Ανεξάρτητα από το σύστημα επεξεργασίας ροών δεδομένων που χρησιμοποιείται ο κάθε χρήστης θέλει να έχει βέλτιστη χρήση των πόρων που έχει στη διάθεση του (η βελτιστοποίηση αυτή μπορεί να εξαρτάται από πολλαπλούς παράγοντες).

Για το Apache Storm συγκεκριμένα, ο χρήστης καλείται να αποφασίσει:

- Τον αριθμό των κόμβων στους οποίους θα τρέξει η εφαρμογή του
- Τον βαθμό παραλληλίας κάθε επεξεργαστικής μονάδας που περιλαμβάνει η εφαρμογή

Τις παραπάνω παραμέτρους δεν είναι απαραίτητο ότι τις γνωρίζει εκ των προτέρων. Αυτό είναι λογικό, αφού πρόκειται για παραμέτρους που έχουν να κάνουν με το runtime. Αποτέλεσμα είναι η εφαρμογή να μην έχει τις αναμενόμενες επιδόσεις.

Η απόφαση που θα πάρει ο χρήστης για τις παραμέτρους παραλληλίας είναι στις περισσότερες περιπτώσεις μη βέλτιστη.

Είναι υποχρεωμένος, λοιπόν, να ακολουθήσει μια διαδικασία trial and fail, μέχρι να βρει την ιδανική παραμετροποίηση της εφαρμογής του. Αυτή η διαδικασία περιλαμβάνει πολλαπλά deploys της εφαρμογής στο σύστημα και παρακολούθηση της συμπεριφοράς κατά τη διάρκεια της εκτέλεσης από τον χρήστη.

1.3.2.2 Στόχος της διπλωματικής

Στόχος είναι η ανάπτυξη ενός τρόπου υποβοήθησης του χρήστη, με στόχο κάθε επιθυμητή εφαρμογή να γίνεται deploy με τις βέλτιστες παραμέτρους παραλληλίας. [2]

Στα πλαίσια της διπλωματικής θα αναπτυχθεί ένα σύστημα αυτόματης εύρεσης βέλτιστου πλάνου παραλληλίας για δεδομένη εφαρμογή.
Το σύστημα αυτό θα στηριχθεί σε μοντελοποίηση των δομικών μονάδων της εφαρμογής μέσω μοντέλων machine learning
Στη συνέχεια μέσω ευριστικών μεθόδων θα καταλήγει στο τελικό βέλτιστο πλάνο

Επιπρόσθετα, θα αναπτυχθεί ένα σύστημα που θα βρίσκει τη λύση στο runtime περιβάλλον και θα προσφέρει elasticity. Είναι μια rule based εφαρμογή που θα της δίνεται ένας στόχος και θα προσπαθεί να τον επιτύχει προσαρμόζοντας κατάλληλα τις παραμέτρους παραλληλίας.
Ουσιαστικά θα παρακολουθεί την εκτέλεση και όποτε κρίνει αναγκαίο θα κάνει τις απαραίτητες αλλαγές για την επίτευξη του στόχου.

1.3.3 Δομή διπλωματικής

Ο τρόπος που θα δομήσουμε τη διπλωματική θα προσφέρει μια εικόνα εις βάθος στο περιβάλλον(Apache Storm) καθώς και στα συστήματα που θα υλοποιήσουμε. Γι' αυτό το λόγο η προσέγγιση θα γίνει σταδιακά και θα περιλαμβάνει τα εξής διακριτά κομμάτια :

- Θα ξεκινήσουμε με την παρουσίαση του περιβάλλοντος Apache Storm, μέσω της ανάλυσης των βασικών εννοιών και δομικών μονάδων που περιλαμβάνει. Στη συνέχεια θα εμβαθύνουμε στην αρχιτεκτονική του συστήματος με στόχο την καλύτερη κατανόηση της εσωτερικής λειτουργίας του. Σκοπός είναι να θέσουμε τη βάση για να γίνει ο τρόπος προσέγγισης του προβλήματος, καθώς και τους μηχανισμούς που θα προτείνουμε για την επίλυση του. (Κεφάλαιο 2)
- Θα συνεχίσουμε με την ανάλυση των μηχανισμών που θα υλοποιήσουμε. Η ανάλυση συμπεριλαμβάνει το θεωρητικό υπόβαθρό καθώς και την αρχιτεκτονική-δομή του κάθε μηχανισμού. (Κεφάλαιο 3)

Τέλος θα προχωρήσουμε σε πειραματική αποτίμηση των μηχανισμών και παρουσίαση των συμπερασμάτων. (Κεφάλαιο 4)

2 Ανάλυση Περιβάλλοντος

Το Apache Storm αποτελεί τη βάση της εργασίας μας. Οπότε, θα πρέπει να γίνουν κατανοητά σε βάθος η αρχιτεκτονική του συστήματος και ο τρόπος λειτουργίας των επιμέρους τμημάτων του.

Παρακάτω θα αναλύσουμε τις σχεδιαστικές παραμέτρους του συστήματος. Η κατανόηση της δομής του Apache Storm είναι ιδιαίτερα σημαντική ώστε να μπορέσουμε να σχεδιάσουμε σωστά την αρχιτεκτονική των επιπρόσθετων συστημάτων για την εύρεση της βέλτιστης παραλληλίας.

2.1 Βασικές έννοιες

[3]

Τοπολογίες (Topologies)

Η λογική για μια εκτελέσιμη εφαρμογή είναι συγκεντρωμένη σε μια τοπολογία. Η τοπολογία στο Storm είναι ουσιαστικά ένας DAG όπως αναλύσαμε στην εισαγωγή. Μια τοπολογία στο Storm είναι το ανάλογο με ένα job στο MapReduce, με την διαφορά ότι η τοπολογία δεν τερματίζει ποτέ αλλά τρέχει συνεχώς. Κάθε τοπολογία αποτελείται από spouts, bolts και streams, ένα παράδειγμα φαίνεται στο σχήμα 2.1

Ροές(Streams)

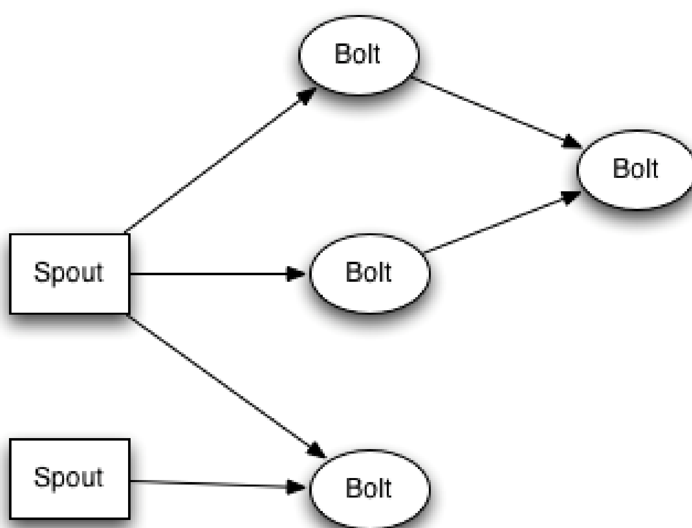
Η ροή στο Storm αποτελείται από συνεχόμενα tuples. Τα tuples είναι μια δομή που περιλαμβάνει τα δεδομένα και μπορεί να περιέχει οποιουδήποτε είδους και μορφής πληροφορία(από integers, bytes arrays, doubles μέχρι custom objects). Κάθε ροή χαρακτηρίζεται από ένα σχήμα(schema) που ονομάζει κάθε πεδίο των tuples της ροής. Επίσης σε κάθε ροή δίνεται ένα id όταν δημιουργείται.

Spouts

Το spout είναι η πηγή των ροών μιας τοπολογίας. Γενικά οι πηγές μεταδίδουν tuples στην τοπολογία από πληροφορία που είτε παράγουν είτε διαβάζουν από κάποιο εξωτερικό σύστημα. Οι πηγές μπορούν να είναι reliable ή unreliable. Ένα reliable spout είναι ικανό να επαναλάβει ένα tuple που απέτυχε να επεξεργαστεί για οποιονδήποτε λόγο. Από την άλλη ένα unreliable spout ξεχνάει την ύπαρξη του tuple κατευθείαν με το που το μεταδώσει. Το reliability θα αναλυθεί σε επόμενο στάδιο.

Bolts

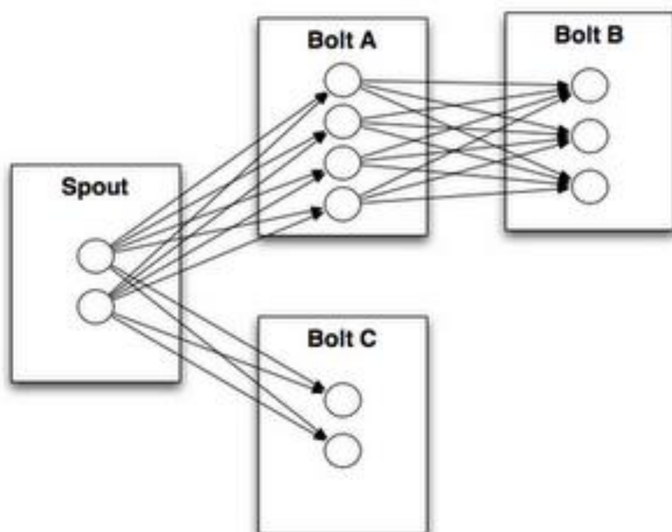
Όλη η επεξεργασία στα bolts γίνεται από τα bolts. Μπορούν να περιέχουν οποιαδήποτε λογική και κατά συνέπεια να εκτελούν οποιαδήποτε λειτουργία από filtering, functions,aggregations,joins μέχρι να μιλάνε με databases. Είναι θεμιτό κάθε bolt να κάνει απλές λειτουργίες πάνω στο stream. Αυτό σημαίνει ότι ένα περίπλοκο stream transformation χρειάζεται περισσότερα επεξεργαστικά στάδια που σημαίνει πολλαπλά bolts. Τα bolts μπορούν να έχουν ως είσοδο και αντίστοιχα να παράγουν πολλαπλές ροές δεδομένων.



Σχήμα 2.1 - Storm topology

Stream grouping

Σημαντικό κομμάτι μιας τοπολογίας είναι η διασύνδεση μεταξύ των spouts και bolts. Πρέπει λοιπόν να οριστεί για κάθε bolt ποια streams θα πρέπει να λαμβάνει σαν είσοδο. Ένα stream grouping είναι ο τρόπος που οι ροές διαμοιράζονται μεταξύ των bolts. Το Storm προσφέρει 8 διαφορετικά stream grouping(τρόπους διασύνδεσης). Παράδειγμα τοπολογίας με τις διασυνδέσεις μεταξύ των spouts και bolts φαίνεται στο σχήμα 2.2



Σχήμα 2.2 - Stream groupings

Επιπλέον υπάρχουν οι παρακάτω οντότητες που έχουν να κάνουν με την εκτέλεση των εφαρμογών το σύστημα:

Worker Process

Εκτελεί ένα υποσύνολο της τοπολογίας και τρέχει στο δικό του JVM. Ένα worker process ανήκει σε μια τοπολογία και μπορεί να τρέχει έναν ή περισσότερους executors για ένα ή περισσότερα components(spouts, bolts) αυτής της τοπολογίας. Μια τοπολογία αποτελείται από πολλά τέτοια processes που τρέχουν σε πολλά μηχανήματα του Storm cluster.

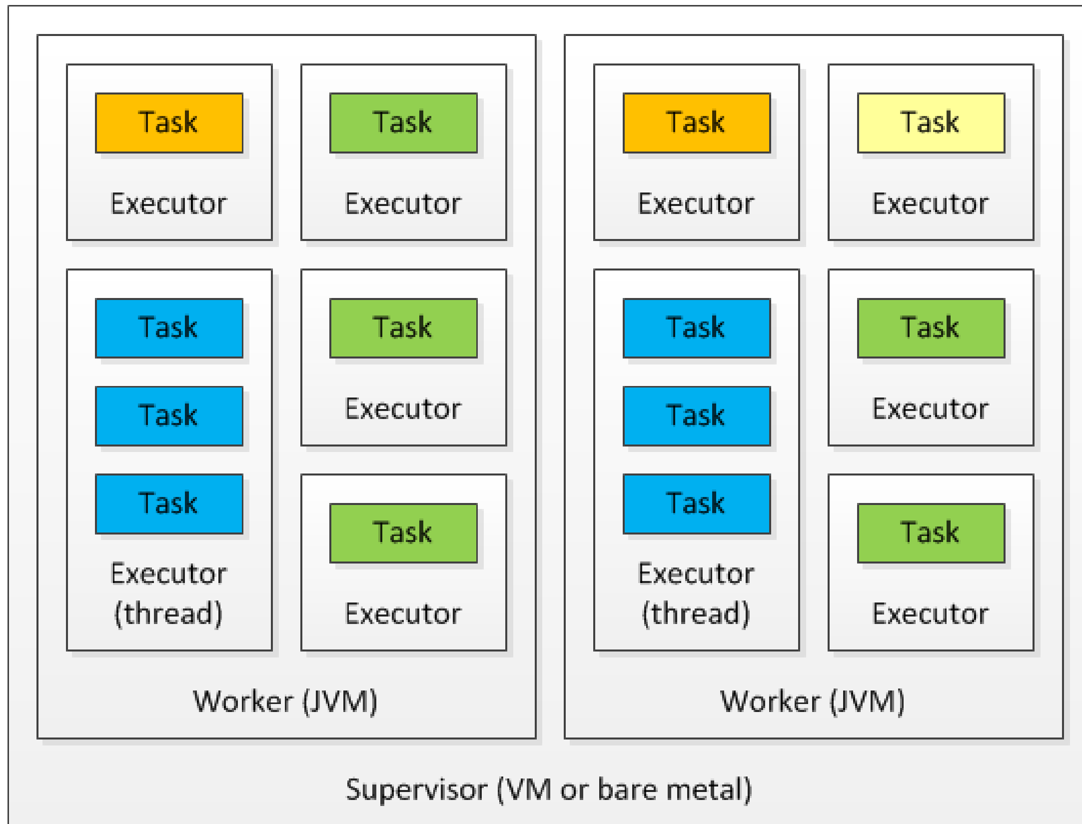
Executor

Πρόκειται για ένα thread που παράγεται από το worker process και τρέχει ένα ή περισσότερα tasks για το ίδιο component(spout, bolt).

Task

Εκτελεί την πραγματική επεξεργασία δεδομένων. Ο αριθμός των tasks για ένα component παραμένει ίδιο καθ' όλη τη διάρκεια ζωής της τοπολογίας, όμως ο

αριθμός των executors(threads) μπορεί να αλλάζει. Αυτό σημαίνει ότι ισχύει ο παρακάτω περιορισμός #threads ≤ #tasks.



Σχήμα 2.3 - Worker structure

2.2 Αρχιτεκτονική του Apache Storm

2.2.1 Σχεδιαστικές επιλογές

2.2.1.1 Δομικές μονάδες

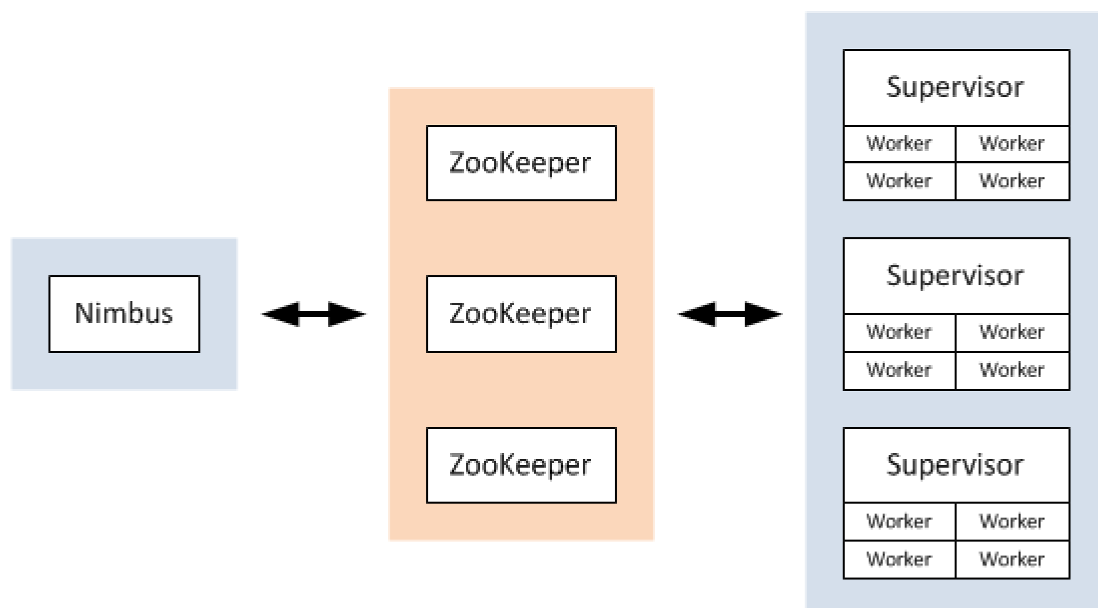
Όπως αναφέραμε και στην εισαγωγή ο δομή του συστήματος είναι ιδιαίτερα σημαντική. Το Apache Storm ακολουθεί το μοντέλο master-slave. Χρησιμοποιεί αυτό το μοντέλο ως βάση και το επεκτείνει για να καλύψει τις αδυναμίες του.

Συγκεκριμένα, το Storm αποτελείται από δύο είδη κόμβων :

- **Master node**. Αυτός ο κόμβος τρέχει το Nimbus daemon που είναι υπεύθυνο να αναθέτει τις εργασίες στους worker nodes, να εκτελεί monitoring στο cluster για τυχόν αποτυχίες, να διαμοιράζει τον κώδικα στο cluster.
- **Worker node**. Κάθε worker node τρέχει το Supervisor daemon. Ο Supervisor ακούει τις κατευθύνσεις από το Nimbus και ξεκινάει ή σταματάει εργασίες ανάλογα.

Σύμφωνα με την παραπάνω αρχιτεκτονική οι worker nodes επικοινωνούν με τον master node μόνο για την ανάθεση της εργασίας και για monitoring σκοπούς. Αυτό ελαφρύνει τη δουλειά του master node και έτσι δεν αποτελεί bottleneck.

Όλος ο συντονισμός μεταξύ Nimbus και Supervisors γίνεται μέσω ενός Apache Zookeeper cluster. [4]
Το Apache Zookeeper προσφέρει συντονισμό κατανεμημένων συστημάτων με υψηλή αξιοπιστία. Είναι μια κεντρική υπηρεσία για maintaining configuration information, naming, distributed synchronization.



Σχήμα 2.4 - Apache Storm Cluster

2.2.1.2 Fault Tolerance

Τα Nimbus daemons και Supervisor daemons είναι fail-fast και stateless.

Το state διατηρείται στο Zookeeper. Αυτό είναι πολύ σημαντικό γιατί αποφεύγεται το λεγόμενο single point of failure στην master-slave αρχιτεκτονική.

Fail-fast σημαίνει ότι οι διεργασίες αυτοκαταστρέφονται όταν συναντήσουν κάποια απρόσμενη κατάσταση)

Κανένα worker process δεν επηρεάζεται από τον θάνατο του Nimbus ή των Supervisors. (Σε αντίθεση με το Hadoop που αν ο JobTracker πεθάνει, χάνονται τα jobs που τρέχουν)

Το να πεθάνει το Nimbus, δηλαδή ουσιαστικά να πεθάνει ο master, δεν επηρεάζει το σύστημά μας και μπορεί να ξεκινήσει εκ νέου χωρίς επιπτώσεις. Βέβαια η εξής συμπεριφορά προκύπτουν κατά τη διάρκεια του Nimbus unavailability:

- Οι υπάρχουσες τοπολογίες συνεχίζουν αν τρέχουν χωρίς να επηρεάζονται, αφού οι Supervisors επικοινωνούν με το Nimbus μόνο όταν τους ανατίθενται καινούριες εργασίες.
- Δεν μπορούν να υποβληθούν στο σύστημα καινούριες τοπολογίες.
- Οι υπάρχουσες τοπολογίες δεν μπορούν να γίνουν kill, deactivate, activate ή rebalance.

Όλα τα παραπάνω εγγυόνται high availability του cluster. [5]

2.2.2 Κατανόηση των εσωτερικών λειτουργιών

2.2.2.1 Internal messaging

Όταν κάποιος προσπαθεί να βελτιστοποιήσει τις τοπολογίες στο Apache Storm είναι χρήσιμο να έχει κατανοήσει πώς είναι σχεδιασμένος ο εσωτερικός μηχανισμός επικοινωνίας. [6]

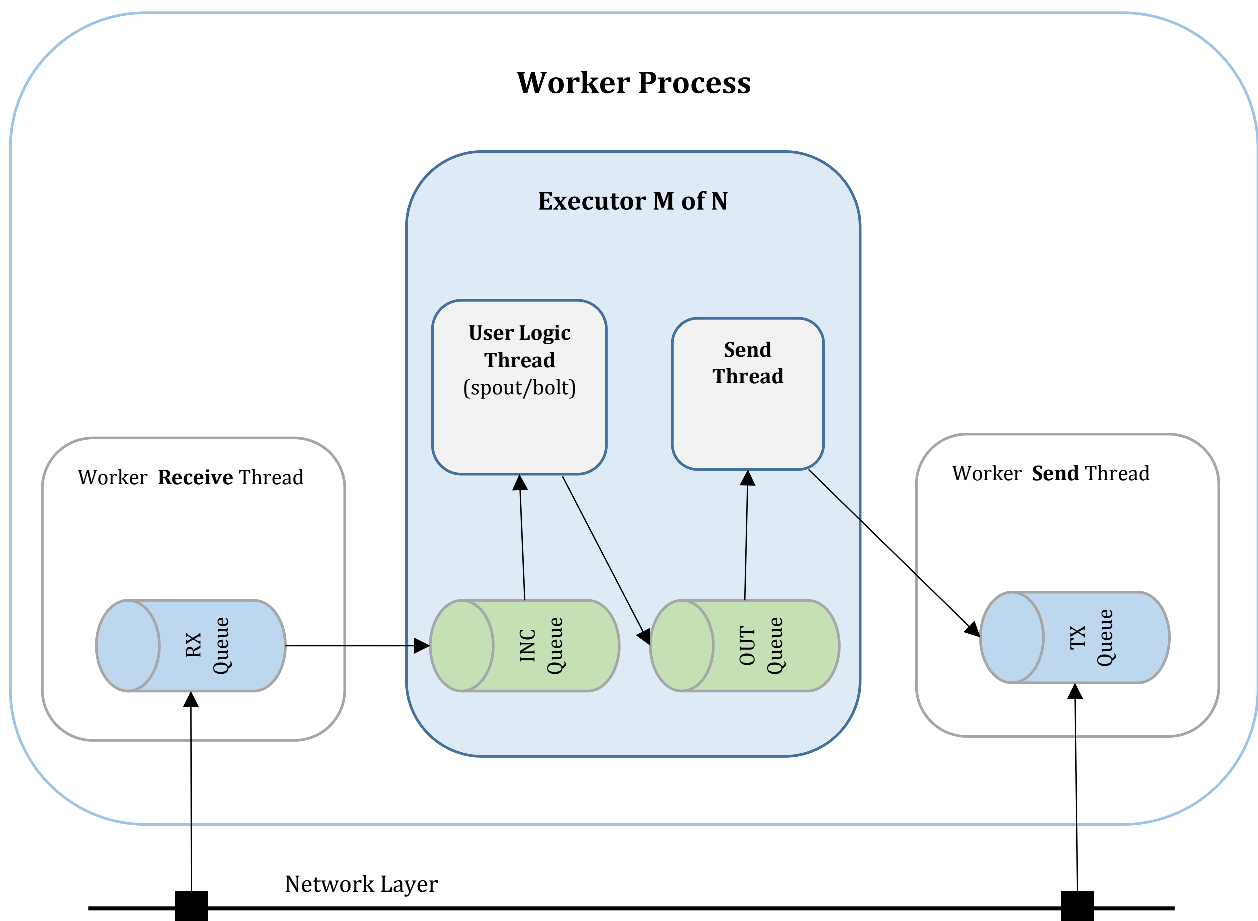
Στην πραγματικότητα υπάρχουν 2 μηχανισμοί επικοινωνίας στο Storm και είναι :

- **Intra-worker communication**. Πρόκειται για την «εσωτερική» επικοινωνία, δηλαδή αυτή που γίνεται στα πλαίσια ενός worker process στο Storm. Αυτή η επικοινωνία είναι περιορισμένη στο ίδιο μηχάνημα και το Storm χρησιμοποιεί message queues του LMAX Disruptor, που πρόκειται για μια βιβλιοθήκη high performance interthread messaging.
- **Inter-worker communication**. Πρόκειται για την επικοινωνία μεταξύ των worker processes και κατά συνέπεια μεταξύ μηχανημάτων που ανήκουν στο cluster. Για τον σκοπό αυτό χρησιμοποιείται το Netty.

Στο σχήμα 2.5 φαίνεται παραστατικά ο τρόπος επικοινωνίας του Storm. Στη συνέχεια θα αναλύσουμε τις διάφορες παραμέτρους τις επικοινωνίας.

Worker Process messaging

Κάθε worker για να διαχειρίζεται τα εισερχόμενα μηνύματα έχει ένα thread που ακούει στην TCP port του worker για μηνύματα από άλλους workers και τα τοποθετεί σε μια ουρά. Στη συνέχεια δρομολογεί τα μηνύματα στις incoming queues των executors.
Με την ίδια λογική, κάθε worker έχει ένα thread που είναι υπεύθυνο για τα εξερχόμενα μηνύματα. Οι executors αφού τελειώσουν την επεξεργασία και έχουν δεδομένα να στείλουν, τοποθετούν τα μηνύματα στην transfer queue του worker. Το send thread διαβάζει από την ουρά και τα στέλνει μέσω του δικτύου στους worker στους οποίους εκτελούνται τα επόμενα επεξεργαστικά στάδια.



Σχήμα 2.5 - Storm messaging

Executors

Κάθε worker process ελέγχει ένα ή περισσότερα executor threads. Κάθε executor thread έχει τα δικά τους incoming και outcoming queues. Το receive thread, όπως περιγράφεται παραπάνω, είναι υπεύθυνο για να μεταφέρει τα εισερχόμενα μηνύματα στα κατάλληλα incoming queues των executor threads του worker. Με παρόμοιο τρόπο, ο κάθε executor έχει ένα send thread που μεταφέρει τα εξερχόμενα μηνύματα του executor από το outgoing queue στο transfer queue του worker.
Έτσι κάθε executor περιέχει ένα thread που διαχειρίζεται την «λογική» του spout/bolt, δηλαδή είναι το thread που αναλαμβάνει την επεξεργασία, και ένα thread που μεταφέρει τα μηνύματα από outgoing queue στην transfer queue.

Συμπερασματικά, είναι πολύ σημαντική η κατανόηση των μηχανισμών επικοινωνίας καθώς και το σωστό configuration αυτών. Επιγραμματικά, σημαντικές παράμετροι είναι οι εξής:

- **topology.receiver.buffer.size** (default = 8) Δείχνει τον μέγιστο αριθμό των μηνυμάτων που γίνονται batch και προστίθενται από το receive thread του worker στην incoming queue του executor. Πρέπει να είναι δύναμη του 2, λόγω της αρχιτεκτονικής του LMAX Disruptor. Δεν πρέπει να τεθεί σε μεγάλα νούμερα γιατί δημιουργούνται προβλήματα(το heartbeat thread γίνεται starve, το throughput πέφτει).
- **topology.transfer.buffer.size** (default = 1024) Δείχνει το μέγεθος της transfer queue του worker. Όλα τα executor threads κάνουν batch τα outgoing tuples σε αυτήν.
- **topology.executor.receive.buffer.size** (default = 1024) Δείχνει το μέγεθος της incoming queue των executors. Κάθε element αυτής της ουράς είναι μια λίστα από tuples γιατί προστίθενται σε batches από το receive thread. Πρέπει να είναι δύναμη του 2.
- **topology.executor.send.buffer.size** (default =1024) Δείχνει το μέγεθος της outgoing queue των executors. Κάθε element αυτής της ουράς είναι ένα tuple. Πρέπει να είναι δύναμη του 2.

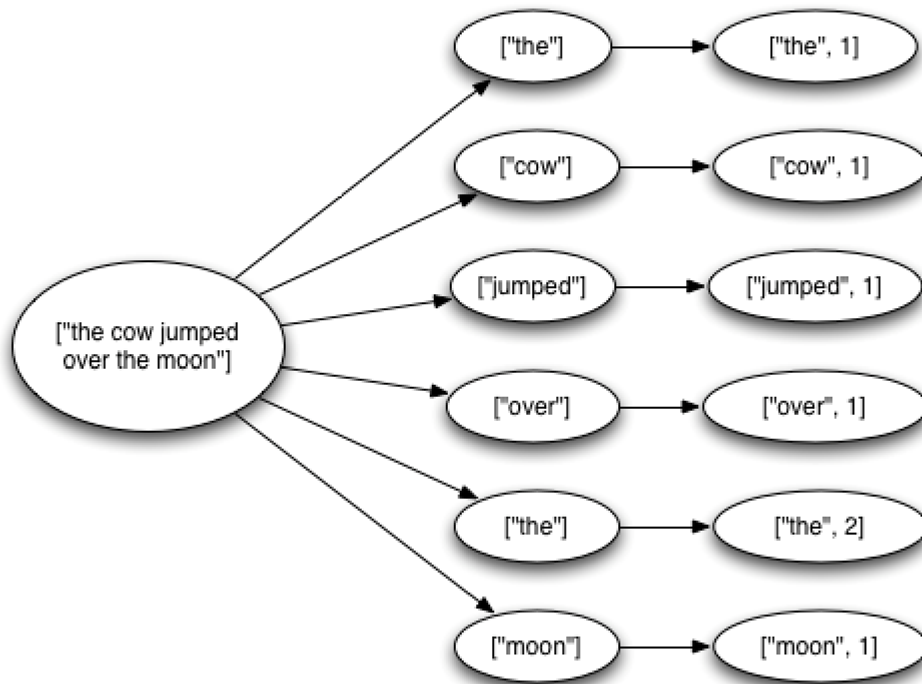
2.2.2.2 Guaranteeing message processing

Το Storm προσφέρει εγγυημένη επεξεργασία των δεδομένων και στη συνέχεια θα περιγράψουμε τον μηχανισμό με τον οποίο παρέχεται. [7]

Ένα tuple που παράγεται από ένα spout μπορεί να οδηγήσει στην δημιουργία δεκάδων, εκατοντάδων ή ακόμη και χιλιάδων άλλων tuples. Για να γίνει κατανοητό το concept του μηχανισμού θα χρησιμοποιήσουμε ως παράδειγμα μια τοπολογία word count. Αποτελείται από ένα spout που παράγει προτάσεις και 2 bolts, ένα που σπάει την πρόταση σε λέξεις και ένα που μετράει τις εμφανίσεις λέξεων. Έτσι ένα tuple που ξεκινάει από το spout και είναι μια πρόταση οδηγεί στη δημιουργία πολλών άλλων tuples αφού σπάσει η πρόταση σε λέξεις και στη συνέχεια μεταφερθεί για να μετρηθούν οι λέξεις. Δημιουργείται ουσιαστικά ένα δέντρο από tuples όπως στο σχήμα 2.6

Το Storm θεωρεί ότι τελείωσε η επεξεργασία ενός spout tuple όταν έχει επεξεργαστεί όλο το tuple tree. Ένα tuple θεωρείται failed αν το tuple tree δεν έχει επεξεργαστεί πλήρως μέσα σε ένα ορισμένο χρονικό διάστημα(timeout).

Κάθε spout έχει ack και fail μεθόδους για να μπορεί ο χρήστης να διαχειριστεί όπως θέλει τα επιτυχημένα ή αποτυχημένα tuples.



Σχήμα 2.6 - Tuple Tree

Αυτή η διαδικασία θα πρέπει να είναι αποδοτική ώστε να μην επιβαρύνεται πολύ η επίδοση του συστήματος. Γι' αυτό το λόγο κάθε τοπολογία στο Storm χρησιμοποιεί ειδικά threads που ονομάζονται "acker", τα οποία κρατάνε την πορεία του tuple tree για κάθε spout tuple και μόλις δουν ότι ολοκληρώθηκε στέλνουν μήνυμα στο spout που το παρήγαγε.

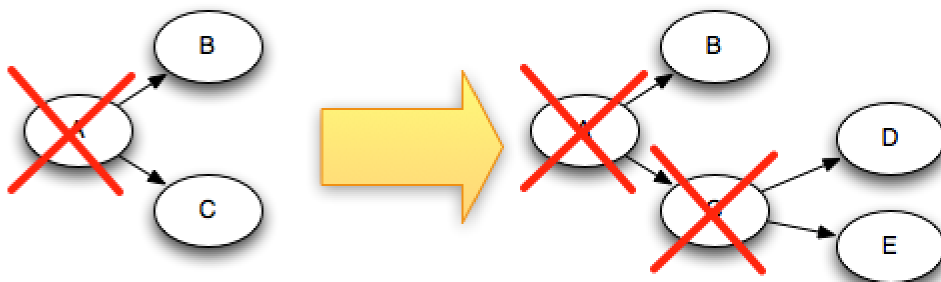
Ένας acker κρατάει ένα map από ένα spout tuple id σε ένα ζευγάρι τιμών. Η πρώτη τιμή είναι το task id που δημιούργησε το spout tuple και χρησιμοποιείται στη συνέχεια για να σταλεί μήνυμα ολοκλήρωσης επεξεργασία πίσω στο spout. Η δεύτερη τιμή είναι ένα αριθμός 64 bit που ονομάζεται ack val. Το ack val είναι μια αναπαράσταση του state του tuple tree ανεξάρτητα από το μέγεθος του. Είναι απλά ένα xor όλων των tuple ids που δημιουργήθηκαν ή/και ολοκληρώθηκαν στο tuple tree. Όταν ο acker δει ότι ένα ack val έχει γίνει 0, ξέρει ότι το αντίστοιχο tuple tree έχει ολοκληρωθεί. Η πιθανότητα να γίνει από τυχαίο λάθος 0 είναι πολύ μικρή γιατί τα tuple ids είναι αριθμοί 64 bit.

Σε κάθε tuple που δημιουργείται στην τοπολογία είτε από spout είτε από bolt δίνεται ένα τυχαίο 64 bit id, το οποίο έπειτα στέλνεται από τ component στον κατάλληλο acker ώστε να αλλάξει η τιμή του ack val, δηλαδή το state του tuple tree.
Έτσι κάθε component(spout, bolt) της τοπολογίας επικοινωνεί με τον acker για να δηλώσει κάποια αλλαγή στο tuple tree είτε για δημιουργία καινούριων tuples είτε για την επιβεβαίωση επεξεργασίας αυτών.

Υπάρχουν, όμως, πολλά acker tasks σε μια τοπολογία και καθένα διαχειρίζεται συγκεκριμένα tuple trees. Πώς θα γνωρίζουν τα components σε πιο να στείλουν; Μαζί με κάθε tuple που περνάει από τα διάφορα components κρατείται μια λίστα από spout tuple ids όλων των tuple trees που ανήκουν και έτσι γνωρίζουν με ποιο ή ποια acker tasks να επικοινωνήσουν.

Ο καλύτερος τρόπος για να γίνει κατανοητός ο μηχανισμός είναι αν δούμε το lifecycle ενός tuple tree.

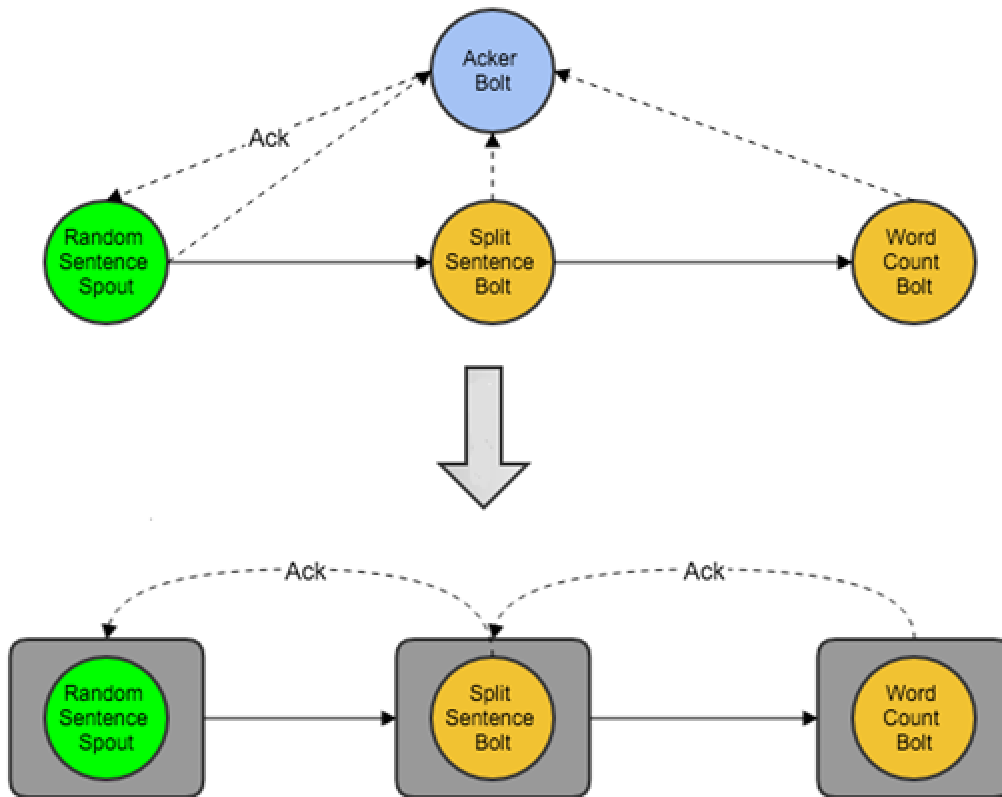
1. Το spout A δημιουργεί ένα spout tuple A με $spout\ tuple\ id\ =\ idA$
2. Το spout που δημιούργησε το A επικοινωνεί με τον acker και του στέλνει το idA
3. Ο acker δημιουργεί μια εγγραφή της μορφής $\{idA, (spoutId, idA)\}$
4. Το tuple A στέλνεται στο bolt B, το οποίο δημιουργεί 2 νέα tuples, tuple B, tuple C.
5. Το bolt B στέλνει στον acker τη δημιουργία των tuples B,C καθώς και την επεξεργασία του A.
6. Ο acker θα ενημερώσει την εγγραφή για το spout tuple A σε $\{idA, (spoutId, idA\ xor\ idA\ xor\ idB\ xor\ idC)\} =$ $\{idA, (spoutId, idB\ xor\ idC)\}$
7. Το tuple C στέλνεται στο bolt C, το οποίο δημιουργεί 2 νέα tuples, tuple D, tuple E.
8. Το bolt B στέλνει στον acker τη δημιουργία των tuples D,E καθώς και την επεξεργασία του C.
9. Ο acker θα ενημερώσει την εγγραφή για το spout tuple C σε $\{idA, (spoutId, idB\ xor\ idC\ xor\ idC\ xor\ idD\ xor\ idE)\} =$ $\{idA, (spoutId, idB\ xor\ idD\ xor\ idE)\}$
10. Μόλις το ack val γίνει 0 από τα αλληλοαναιρούμενα xor τότε θα σταλεί μήνυμα από το acker στο spout A για την ολοκλήρωση του tuple A
11. Αν το ack val δεν γίνει ποτέ 0 μέσα σε ένα προκαθορισμένο χρονικό διάστημα, ο acker θα στείλει μήνυμα στο spout A για αποτυχία του tuple A



Σχήμα 2.7 - Tuple Tree Lifecycle

Στο σχήμα 2.7 φαίνεται ο τρόπος με τον οποίο προχωράει το tuple tree καθώς επεξεργάζονται τα tuples και δημιουργούνται νέα από τα διάφορα components.

Στο σχήμα 2.8 φαίνεται η τοπολογία Word Count και ο τρόπος με τον οποίο γίνεται η επικοινωνία μεταξύ των components και του acker.



Σχήμα 2.8 - Component-Acker Communication

Τα acker bolts μπορούν να είναι αυθαίρετα πολλά. Όμως επειδή εκτελούν μια ελαφριά διαδικασία δεν χρειάζονται πολλά σε μια τοπολογία. Ο ιδανικός αριθμός είναι 1 ανά worker process για να πετυχαίνουμε καλύτερο locality.

Σε περιπτώσεις που παράγεται μεγάλος αριθμός tuples μπορεί να λάβουν μεγάλο μέρος των πόρων του συστήματος και θα πρέπει να λαμβάνεται σοβαρά υπόψιν αν είναι απαραίτητο να υπάρχει tuple replay όταν το throughput είναι υψηλό.

Τέλος είναι εμφανές ότι οι επιβεβαιώσεις που στέλνονται για την επεξεργασία των διάφορων tuples λαμβάνουν μεγάλο μέρος του network. Συγκεκριμένα αν απενεργοποιηθεί το tracking των tuple trees τα μηνύματα που μεταφέρονται υποδιπλασιάζονται, αποτέλεσμα λογικό αφού στέλνεται ένα ack μήνυμα για κάθε tuple του tuple tree. Επιπροσθέτως, δεν απαιτείται να κρατούνται ids πάνω στα tuples πράγμα που μειώνει το bandwidth usage.

Είναι σημαντικό να αποφασίζεται για κάθε τοπολογία αν είναι απαραίτητη η χρήση του μηχανισμού. Αν είναι θα πρέπει να γίνουν συμβιβασμοί στις επιδόσεις του συστήματος.

2.2.2.3 Parallelization

Τέλος θα αναλύσουμε τον τρόπο με το οποίο λειτουργεί η παραλληλία στο Storm και επιτρέπει στις τοπολογίες να κλιμακώνει. [8]

Η παραλληλία έχει να κάνει με τον τρόπο που εκτελείται η τοπολογία στο Storm και οι οντότητες που σχετίζονται με το runtime αναλύθηκαν παραπάνω και είναι worker processes, executors(threads), tasks. Συνοπτικά, τα executor threads τρέχουν ένα ή περισσότερα tasks για την επεξεργασία των δεδομένων, ένα worker process είναι ένα JVM και αναλαμβάνει ένα υποσύνολο των executors.

Έτσι ο κάθε χρήστης μπορεί να ορίσει την παραλληλία μεταβάλλοντας τα εξής:

- **Number of worker processes.** Πόσα worker processes θα δημιουργηθούν για την τοπολογία στο Storm cluster.
- **Number of executors(threads).** Πόσοι executors θα δημιουργηθούν για κάθε component(spout/bolt) της τοπολογίας.
- **Number of tasks.** Πόσα tasks θα δημιουργηθούν για κάθε component. Αν δεν οριστεί δημιουργείται ένα task για κάθε executor.

Από τις παραπάνω μεταβλητές ο αριθμός των workers και των executors μπορεί να αλλάξει δυναμικά, δηλαδή αφού έχει γίνει submit η τοπολογία. Αντίθετα, ο αριθμός τον tasks είναι στατικός και δεν μπορεί να αλλάξει, τον λόγο θα τον εξηγήσουμε παρακάτω

```
topologyBuilder.setBolt("downstream-bolt", new GeneralNumberbolt(), 2)
               .setNumTasks(4)
               .shuffleGrouping("upstream-spout");
```

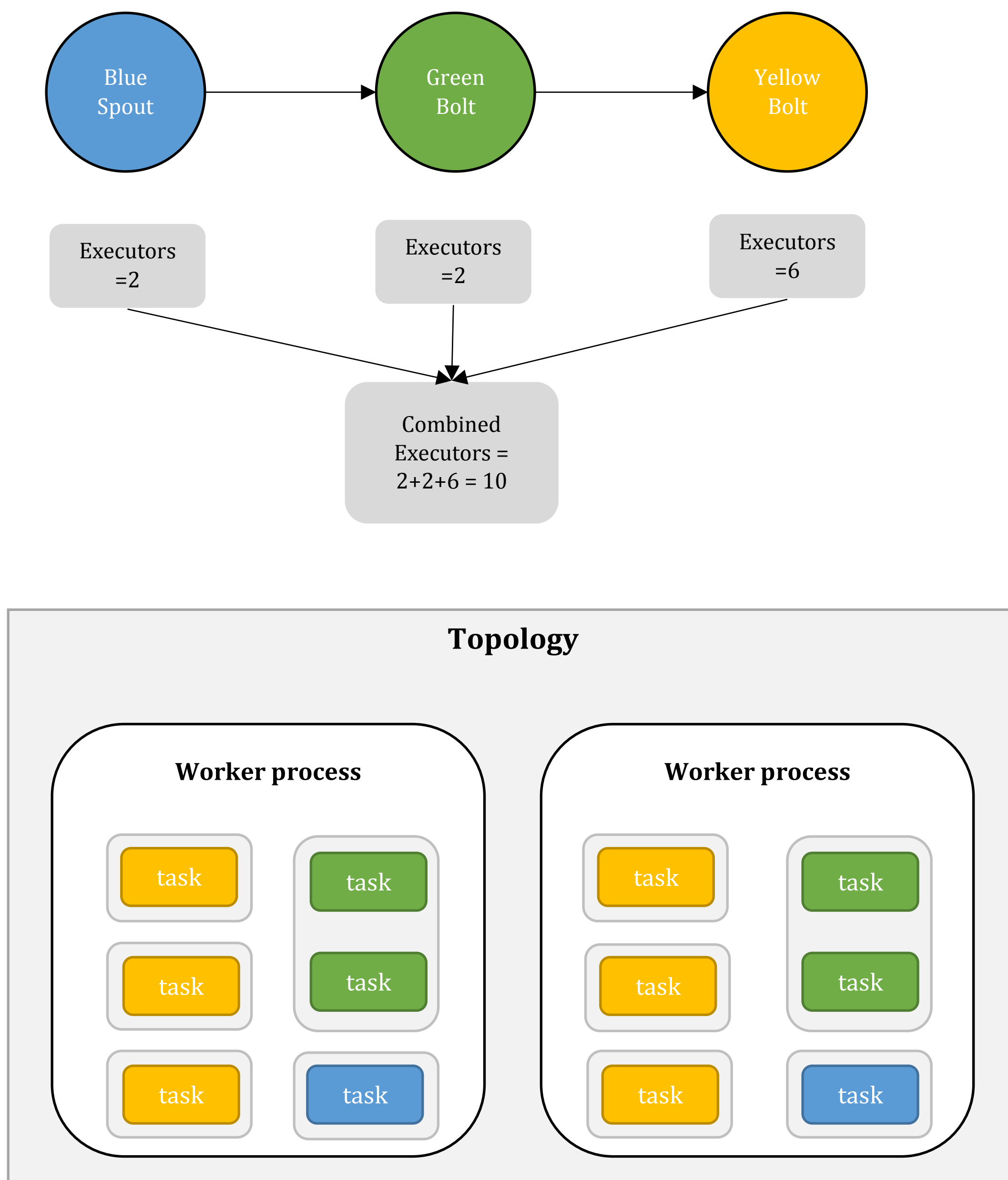
Ο παραπάνω κώδικας δείχνει με παραστατικό τρόπο πως μπορούμε να μεταβάλουμε την παραλληλίας ενός component. Συγκεκριμένα, ορίσαμε το bolt GeneralNumberBolt να έχει 2 executors και 4 tasks. Αυτό σημαίνει ότι θα τρέξουν 2 tasks ανά executor. Όπως αναφέρθηκε και προηγουμένως, αν δεν ορίζαμε τον αριθμό των tasks, by default θα έτρεχε 1 task σε κάθε executor.

Αφού αναλύσαμε την παραλληλία σε ένα component στη συνέχεια θα προχωρήσουμε στην παραλληλία σε μια τοπολογία. Ένα παράδειγμα τοπολογίας σε κώδικα είναι :

```
Config conf = new Config();
conf.setNumWorkers(2);

TopologyBuilder topologyBuilder = new TopologyBuilder();
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
        .setNumTasks(4)
        .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
        .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology("topology1", conf,
        topologyBuilder.createTopology());
```

Συνοπτικά, με τον παραπάνω κώδικα δημιουργήσαμε μια τοπολογία με 1 spout(BlueSpout) και 2 bolts(GreenBolt,YellowBolt). Όσο για την παραλληλία θέσαμε:

- workers = 2
- blue spout → executors = 2 | tasks = 2(default)
- green bolt → executors = 2 | tasks = 4
- yellow bolt → executors = 6 | tasks = 6(default)

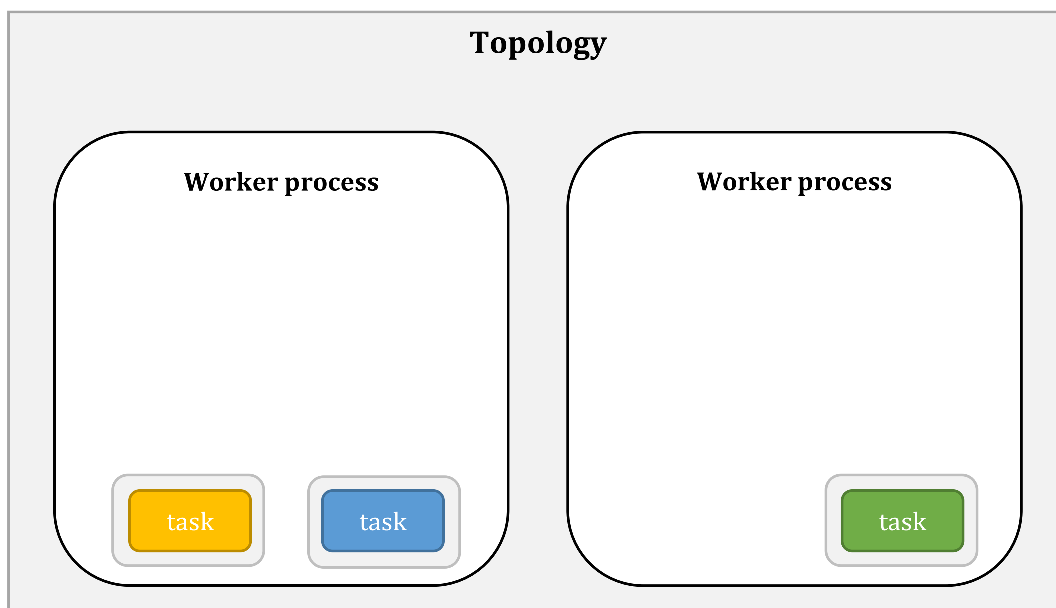


Σχήμα 2.9 - Running Topology 1

Στο σχήμα 2.9 φαίνεται ο τρόπος με τον οποίο θα γίνει deploy και θα τρέξει στο Storm cluster η τοπολογία που δημιουργήσαμε παραπάνω. Συνολικά έχουμε 10 threads που διαμοιράζονται στους 2 workers. Τα components BlueSpout και YellowBolt τρέχουν με default tasks γι' αυτό έχουν και από ένα task σε κάθε executor. Το component GreenBolt έχει 4 tasks που διαμοιράζονται μεταξύ των executors και έτσι έχουμε 2 tasks ανά executor.

Αν όμως είχαμε ακολουθήσει διαφορετική προσέγγιση για την παραλληλία πως θα ήταν η μορφή του cluster; Παρακάτω θα δώσουμε άλλους δύο συνδυασμούς για να γίνει εμφανής τόσο η παραλληλία όσο και ο τρόπος τοποθέτησης των executors στους workers

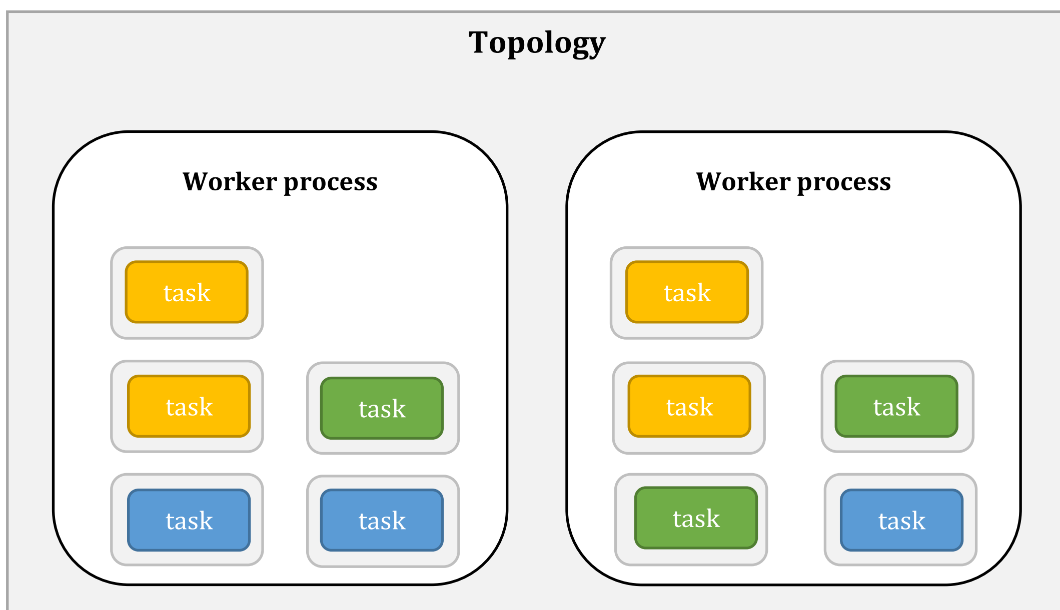
```
Config conf = new Config();
conf.setNumWorkers(2);

TopologyBuilder topologyBuilder = new TopologyBuilder();
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 1);

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 1)
        .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 1)
        .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology("topology2", conf,
        topologyBuilder.createTopology());
```



Σχήμα 2.10 - Running Topology 2

Για την τοπολογία 2 δεν ορίσαμε παραλληλία σε επίπεδο component. Όλα τα components εκτελούνται από 1 executor και κάθε executor έχει 1 task (αφού δεν ορίστηκε το default είναι 1).

```java
Config conf = new Config();
conf.setNumWorkers(2);

TopologyBuilder topologyBuilder = new TopologyBuilder();
topologyBuilder.setSpout("blue-spout", new BlueSpout(), 3);

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 3)
        .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 4)
        .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology("topology3", conf,
        topologyBuilder.createTopology());
```



Σχήμα 2.11 - Running Topology 3

Για την τοπολογία 3 ορίσαμε παραλληλία μόνο σε επίπεδο executor threads.

Είναι προφανές ότι τα 3 παραπάνω configurations παρόλο που κάνουν την ίδια δουλειά δεν θα έχουν τις ίδιες επιδόσεις. Αυτό είναι λογικό αφού συμβάλουν πολλοί παράγοντες, τους οποίους θα αναλύσουμε παρακάτω.

Executors/Worker

Στις τοπολογίες 1,3 κάθε worker process έχει αναλάβει αρκετά threads προς εκτέλεση(6 και 5 executors αντίστοιχα), ενώ η τοπολογία 2 έχει μόλις 1-2 threads

σε κάθε worker. Αυτό δεν είναι απαραίτητα αποδοτικό, ειδικά αν το hardware του machine δεν έχει την δυνατότητα να υποστηρίξει τόσο μεγάλο φόρτο εργασίας. Ακόμα και 1 executor thread μπορεί να κάνει χρήση του συνόλου των πόρων του συστήματος σε περιπτώσεις μεγάλου φόρτου εργασίας.

Έτσι, αν έχουμε executors που χρησιμοποιούν μεγάλο μέρος του cpu time δεν κερδίζουμε τίποτα με την προσθήκη επιπλέον threads. Για να γίνει κατανοητή η συμπεριφορά του συστήματος σε περιπτώσεις που σε ένα worker έχουμε πολλά executor threads θα διακρίνουμε 2 περιπτώσεις :

1. Αν έχουμε πολλά executor threads του ίδιου component, η δουλειά θα μοιραστεί σε αυτά τα threads αλλά η χρήση των πόρων του συστήματος θα είναι η ίδια και όπως με το να είχαμε 1 executor thread. Συνέπεια, εφόσον έχουμε τους ίδιους πόρους για περισσότερα threads η επίδοση δεν βελτιώνεται και υπάρχει περίπτωση αν έχουμε μεγάλο αριθμό threads κάποια να μείνουν αχρησιμοποίητα(idle). Επίσης, όσο περισσότερα threads προσθέτουμε αυξάνεται ο ανταγωνισμός για τους πόρους του συστήματος και έχουμε αυξημένο χρόνο εκτέλεσης λόγω context switching.
2. Αν έχουμε πολλά executor thread και είναι διαφορετικού είδους, έχουμε μεγάλη πιθανότητα να υπάρχουν threads από γειτονικά components(π.χ. τοπολογίες 1,3) και κατά συνέπεια να έχουμε τα δεδομένα που χρειάζονται επεξεργασία κατευθείαν(data locality) στον worker χωρίς να περάσουν από το network. Υπάρχουν όμως και μειονεκτήματα. Αν ένα από τα threads χρειάζεται να επεξεργάζεται δεδομένα πιο συχνά από τα υπόλοιπα (και ειδικά αν έχει και μικρότερο execute time) θα διακόπτει συνεχώς την επεξεργασία των υπόλοιπων(context switch) με αποτέλεσμα να δημιουργούνται καθυστερήσεις. Μάλιστα, αν κάποιο thread λαμβάνει πολύ μεγάλο μέρος των πόρων του worker μπορεί να οδηγήσει σε starvation των υπόλοιπων threads.

Εκτός, όμως, από την υπέρμετρη χρήση των πόρων υπάρχει και το αντίθετο, δηλαδή τα executors να είναι τοποθετημένα με τέτοιο τρόπο που να μην αξιοποιείται το σύνολο των πόρων. Για παράδειγμα στην τοπολογία 2 έχουμε 2 executors στο πρώτο worker και 1 στον δεύτερο, αν υποθέσουμε ότι όλα επεξεργάζονται το ίδιο πλήθος δεδομένων και ότι χρησιμοποιούν το ίδιο κομμάτι πόρων ανά δεδομένο επεξεργασίας τότε ο πρώτος worker θα φτάσει στο 100% χρήσης των πόρων ενώ ο δεύτερος μόλις στο 50%.

Messaging

Τη λογική της εσωτερικής επικοινωνίας του Storm την αναλύσαμε σε προηγούμενο κεφάλαιο. Σε αυτό το σημείο θα επισημάνουμε πως επηρεάζει τις επιδόσεις της κάθε τοπολογίας.

Αν έχουμε πολλά executors του ίδιου είδους σε έναν worker μπορούμε να κερδίσουμε στο internal messaging. Συγκεκριμένα, θα έχουμε περισσότερα

executors του ίδιου είδους, και κατά συνέπεια περισσότερα incoming message buffers που θα μπορούν να παίρνουν πιο συχνά batches από το receive queue του worker. Αυτό θα έχει ως αποτέλεσμα χαμηλότερο latency στην τοπολογία, όχι όμως καλύτερες επιδόσεις(το throughput θα μείνει το ίδιο).

Αν έχουμε πολλά executors από διαφορετικά components σε έναν worker, μπορούμε να κερδίσουμε αν αυτά τα components συνδέονται μεταξύ τους. Σε τέτοια περίπτωση τα δεδομένα που πρέπει να σταλούν δεν χρειάζεται να τοποθετηθούν στον transfer buffer, αλλά τοποθετούνται κατευθείαν από τον send στον receive buffer. Αυτό είναι μεγάλο πλεονέκτημα από άποψη network.

Τέλος, αν έχουμε περισσότερα από ένα workers και πολλά executors που στέλνουν δεδομένα σε πολλούς διαφορετικούς workers, το network time αυξάνεται σημαντικά και επιβαρύνεται συνολικά το σύστημα(flow control). Αποτέλεσμα είναι, όσο αυξάνουμε τους workers να μην βλέπουμε γραμμική αύξηση των επιδόσεων αλλά πτώση στο κέρδος από την προσθήκη workers.

Tasks

Η προσθήκη περισσότερων του ενός task ανά executor δεν αυξάνει την παραλληλία. Ένας executor τρέχει σε ένα thread το οποίο χρησιμοποιούν όλα τα tasks, που σημαίνει ότι τα tasks τρέχουν σειριακά σε έναν executor.

Ο λόγος για να έχει περισσότερα από 1 tasks ανά executor thread είναι για να δώσει κανείς την ευελιξία διεύρυνσης της τοπολογίας δυναμικά. Συγκεκριμένα, αν επιθυμεί ο χρήστης μπορεί να δώσει στην τοπολογία περισσότερα executors για τα components ή και επιπλέον workers. Τα υπάρχοντα επιπλέον tasks θα μοιραστούν στα executors που προστέθηκαν και η τοπολογία θα συνεχίσει τη λειτουργία της. Να σημειωθεί, ότι θα πρέπει πάντα να ισχύει #executors ≤ #tasks.

Άρα τα tasks είναι ένα εργαλείο για να παρέχεται δυναμική προσθήκη πόρων στην τοπολογία.

Workers

Ο αριθμός των workers που χρησιμοποιήσαμε και στις 3 τοπολογίες είναι ίδιος. Αν προσθέταμε περισσότερα worker processes η τοποθέτηση των executor θα ήταν προφανώς διαφορετική και η τοπολογία θα είχε διαφορετική συμπεριφορά σύμφωνα με αυτά που είπαμε και παραπάνω.

Όποιος αριθμός worker και να χρησιμοποιηθεί, είναι προτιμότερο να αυτοί να ανήκουν σε διαφορετικά μηχανήματα. Υπάρχουν 3 λόγοι για τους οποίους θα πρέπει να τρέχουμε 1 worker process ανά machine. Αρχικά, αν έχουμε περισσότερα workers και ανήκουν στο ίδιο machine, executors που προορίζονταν για 1 worker μοιράζονται σε 2 με αποτέλεσμα να χάνουμε το data locality και τα δεδομένα να πρέπει να περάσουν από το network. Η διαδρομή θα είναι ως εξής

send -> worker transfer > local socket -> worker receive > exec receive buffer, δεν πρόκειται αν περάσει από την κάρτα δικτύου αλλά και πάλι το Overhead είναι σημαντικό. Δεύτερον, είναι καλύτερο να κρατάμε λίγους και μεγάλους workers για να γίνεται καλύτερο data caching και να βελτιώνεται το LRU efficiency. Τέλος, με λιγότερους workers έχουμε λιγότερη κίνηση στο δίκτυο λόγω ελέγχου επικοινωνίας.

Συνοπτικά μπορούμε να σημειώσουμε τα εξής όταν προσπαθούμε να βρούμε το βέλτιστο configuration για την κάθε τοπολογία :

- Εκτελούμε 1 worker ανά machine για να κερδίσουμε network overhead και data caching.
- Αν έχουμε μεγάλο αριθμό workers στην τοπολογία αυξάνεται η κίνηση στο δίκτυο και μειώνεται το κέρδος που παίρνουμε στην επίδοση.
- Αν έχουμε γειτονικά executor threads στον ίδιο worker κερδίζουμε network transfer time και data locality.
- Αν έχουμε πολλά executors σε έναν worker υπάρχει κίνδυνος για δημιουργία καθυστερήσεων λόγω context switching.
- Αν έχουμε μεγάλη διασπορά των executors στα workers μπορεί να μην αξιοποιούνται επαρκώς οι πόροι του συστήματος.
- Η τοποθέτηση των executors στους workers γίνεται από τον scheduler. Στο Storm by default ο scheduler είναι round-robin.
- Ο αριθμός των tasks δεν παίζει ρόλο στην παραλληλία.

Κάθε τοπολογία στο Storm cluster μπορεί να αλλάξει παραλληλία δυναμικά. Αυτό ονομάζεται rebalance και μπορούν να προστεθούν ή να αφαιρεθούν workers ή/και executors.

2.3 Ορισμός προβλήματος

Η ανάπτυξη εφαρμογών είναι σχετικά εύκολη υπόθεση στο Storm. Η λογική της επεξεργασίας των δεδομένων σπάει σε μικρότερες επεξεργαστικές μονάδες οι οποίες συνδέονται μεταξύ τους με ροές δεδομένων με τον ίδιο τρόπο που θα έτρεχε η σταδιακή επεξεργασία των δεδομένων αν ήταν ένα ενιαίο κομμάτι.

Το δύσκολο κομμάτι είναι να αποφασίσει τις κατάλληλες παραμέτρους παραλληλίας(workers/executors) για την τοπολογία.

Από την παραπάνω ανάλυση είναι φανερό ότι η εύρεση του βέλτιστου configuration είναι απαιτητική διαδικασία καθώς κανείς πρέπει να λάβει υπόψιν του αρκετές παραμέτρους, μεταξύ των οποίων τις απαιτήσεις κάθε component της τοπολογίας σε πόρους συστήματος και δίκτυο. Συγκεκριμένα, κανείς μπορεί να θεωρήσει ότι οι παράμετροι του παραπάνω προβλήματος βελτιστοποίησης είναι η παραλληλία στην τοπολογία(workers) και η παραλληλία σε κάθε

component(executors). Η πολυπλοκότητα αυξάνεται εκθετικά καθώς προστίθενται components στην τοπολογία.

Ο χρήστης σε αυτήν την περίπτωση δεν έχει πολλές επιλογές καθώς δεν ξέρει την συμπεριφορά των component κατά τη διάρκεια του development. Θα πρέπει να δοκιμάσει πιθανά configuration της τοπολογίας για να βρει μια λύση που τον ικανοποιεί. Αυτό, όμως, δεν αποτελεί εγγύηση ότι αξιοποιεί πλήρως τους πόρους που έχει στη διάθεση του και θα μπορούσε να πετύχει καλύτερες επιδόσεις ή ότι δεν θα μπορούσε να πετύχει τις ίδιες επιδόσεις με μικρότερο κόστος.

Θα προτείνουμε 2 λύσεις στο παραπάνω πρόβλημα βελτιστοποίησης.

Η πρώτη λύση είναι ένας μηχανισμός που προσομοιάζει τον τρόπο με τον οποίο θα έψαχνε και ο χρήστης. Είναι online και rule based. Online με την έννοια ότι λειτουργεί καθώς τρέχει η τοπολογία και παρακολουθεί τις μετρικές της. Rule based με την έννοια ότι για να πετύχει τον στόχο που του δίνεται επιλέγει πως θα κινηθεί με βάση προκαθορισμένους κανόνες.

Η δεύτερη λύση είναι ένας μηχανισμός πρόβλεψης της βέλτιστης λύσης για δεδομένη τοπολογία. Είναι offline , machine learning heuristic. Offline με την έννοια ότι προβλέπει το βέλτιστο configuration για την τοπολογία, αλλά στη συνέχεια δεν παρακολουθεί τον τρόπο με τον οποίο εκτελείται τελικά η τοπολογία. Machine learning και Heuristic με την έννοια ότι χρησιμοποιεί προηγούμενη γνώση για τα components της τοπολογία μέσω μοντέλων, τα οποία στη συνέχεια χρησιμοποιεί για να ψάξει στον χώρο των πιθανών configuration και να βρει τη βέλτιστη λύση.

Και οι δύο παραπάνω μηχανισμοί έχουν στόχο να απαλλάξουν τον χρήστη από το βάρος της βελτιστοποίησης της τοπολογίας προσφέροντας αυτόματη παραλληλοποίηση της εφαρμογής του.

3 Αρχιτεκτονική Συστήματος

3.1 Monitoring

Κοινή βάση και για τους 2 μηχανισμούς αυτόματης παραλληλοποίησης είναι η πληροφορία για τις επιδόσεις των component της τοπολογίας. Αυτή η πληροφορία έρχεται με τη μορφή μετρικών πάνω σε μία running τοπολογία που μας ενδιαφέρει.

Το Storm παρέχει βασικές μετρικές για κάθε component(spout/bolt) μιας τοπολογίας. Αυτές είναι :

Spout

Acked
Ο αριθμός των επιτυχημένων tuple trees.

Failed
Ο αριθμός των αποτυχημένων tuple trees.

Complete Latency
Ο μέσος χρόνος που χρειάζεται ένα tuple tree για να ολοκληρωθεί, δηλαδή με άλλα λόγια ο χρόνος που απαιτείται για ένα tuple που ξεκινάει από το spout να περάσει από το σύνολο της τοπολογίας και να ολοκληρώσει την επεξεργασία του.

Οι παραπάνω τρείς μετρικές είναι διαθέσιμες μόνο για τοπολογίες που έχουν ενεργοποιημένο τον μηχανισμό επιβεβαίωσης επεξεργασίας. Σε αντίθετη περίπτωση είναι 0.

Bolt

Executed
Ο αριθμός των tuples που έχουν επεξεργαστεί.

Acked
Ο αριθμός των tuples που έγιναν ack.

Failed
Ο αριθμός των tuples που έγιναν fail.

Execute Latency
Ο μέσος χρόνος που απαιτείται για να επεξεργαστεί ένα tuple. Αυτός είναι ο χρόνος που παίρνει η execute method του bolt για κάθε εισερχόμενο tuple και μπορεί να είναι χωρίς να σταλεί Ack για το tuple.

Process Latency
Ο μέσος χρόνος που απαιτείται για ένα tuple από της στιγμή που θα το λάβει το bolt(θα κληθεί η execute method) μέχρι να στείλει Ack. Για παράδειγμα bolt που εκτελούν joins, aggregations ή batching μπορεί να μην στείλουν Ack μέχρι να λάβουν κάποια επόμενα tuples.

Capacity
Παίρνει τιμές μεταξύ (0,1) και δείχνει το φόρτο εργασίας. Υπολογίζεται ως $(Executed * Execute\ Latency)/Measurement\ Time$. Αν είναι κοντά στο 1 το bolt δουλεύει στο μέγιστο και πιθανόν να χρειάζεται να αυξηθεί η παραλληλία.

Common

Emitted
Ο αριθμός των tuples που γίνονται emit από το bolt, δηλαδή πόσες φορές καλείται από το bolt η emit method.

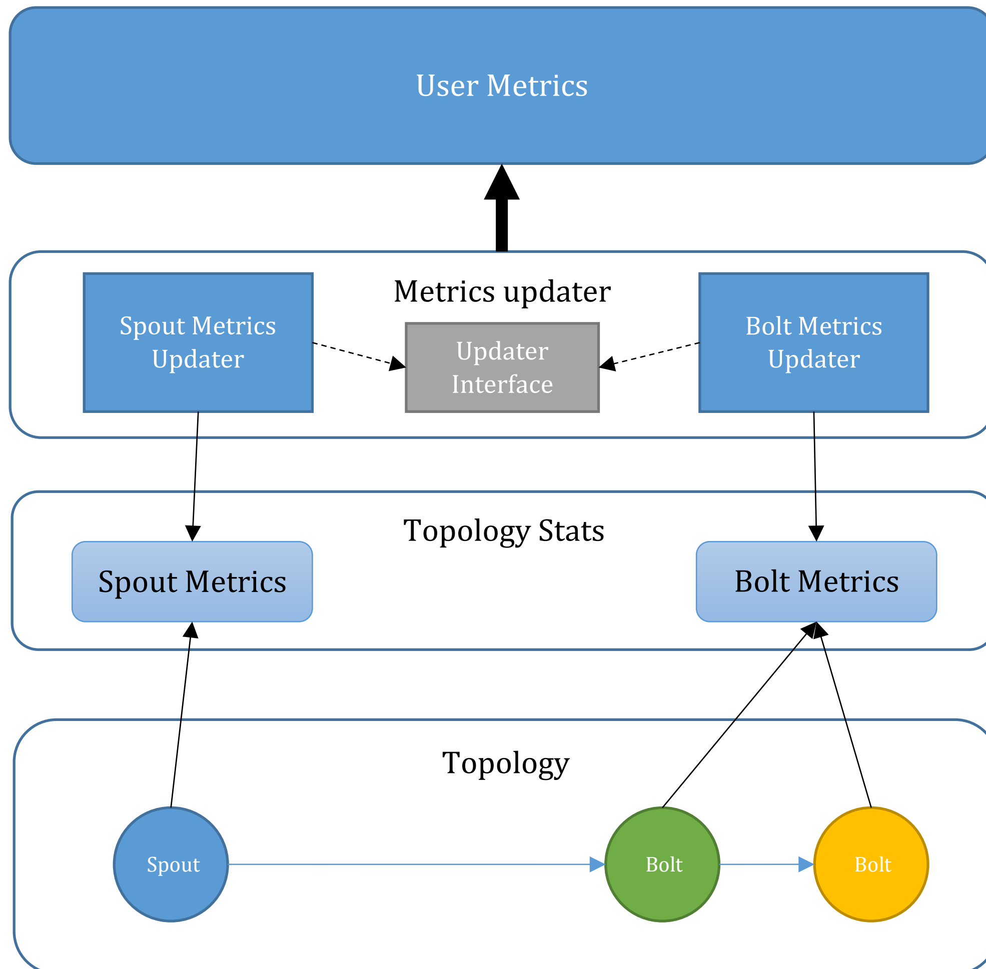
Transferred
Ο αριθμός των tuples που στάλθηκαν σε άλλα components.

Για να γίνει κατανοητή η διαφορά των 2 παραπάνω, αν έχουμε 2 bolts A -> B. Αν το B έχει 5 tasks και ενώνεται με το A με τη μορφή replicated streams(all grouping), στο bolt A θα έχουμε transferred = 5x emitted.

Ο τρόπος με τον οποίο το Storm λαμβάνει αυτές τις μετρικές είναι μέσω δειγματοληψίας. Συγκεκριμένα αν έχουμε δειγματοληψία 5% (default), θα πάρει τυχαία ένα event από τα επόμενα 20 και θα το αυξήσει κατά 20. Αυτό δημιουργεί αβεβαιότητα στις μετρικές, ειδικά όταν οι μετρήσεις διαφέρουν πολύ μεταξύ τους(αν είναι uniform ο τρόπος συμπεριφοράς ακόμα και με μικρή δειγματοληψία παίρνουμε ικανοποιητικά αποτελέσματα). Κανείς μπορεί να θέσει τη δειγματοληψία στο 100% και αν γίνεται monitor κάθε event, αλλά με μεγάλο κόστος στην επίδοση.

Από αυτές τις παραπάνω βασικές μετρικές μπορούμε να χτίσουμε δικές μας που θα μας βοηθήσουν για τους μηχανισμούς που θέλουμε να υλοποιήσουμε. Ο τρόπος με τον οποίο το πετυχαίνουμε φαίνεται στο σχήμα 3.1 και είναι αρκετά ευθύς.

Το επίπεδο του updater συνδέεται με το Storm cluster για να ακούσει τις μετρικές που παρέχει για τα components της τοπολογίας. Ο updater περιλαμβάνει τη λογική με την οποία επεξεργάζονται οι μετρικές που παίρνουμε από το Storm και στη συνέχεια μετατρέπονται στις δικές μας μετρικές. Έχουμε διαφορετικό updater για spouts και bolts καθώς το Storm παρέχει διαφορετικού είδους μετρικές για το κάθε component αλλά και η λογική με την οποία θέλουμε να αντιστοιχιστούν στις δικές μας μετρικές είναι διαφορετική για spouts/bolts. Έτσι, ο updater είναι αυτός που κάνει το mapping Storm metrics – User metrics.



Σχήμα 3.1 - Metrics Architecture

User Metrics

Οι μετρικές που μας δίνει το Storm μπορεί να αφορούν τα components (spouts/bolts), όμως είναι σε επίπεδο executor. Κάθε component μπορεί να έχει περισσότερα από 1 executors. Έτσι πρέπει να συγκεντρώσουμε τις μετρικές για κάθε executor του καθενός component.

Επειδή μεταξύ των executors δεν υπάρχει πάντα ίσα μοιρασμένος φόρτος εργασίας, πολλές από τις μετρικές μας χρησιμοποιούν βεβαρημένους μέσους όρους.

Για ένα μη κενό σύνολο $\{x_1, x_2, \dots, x_n\}$ ο βεβαρημένος μέσος όρος ορίζεται ως:

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}, w_i \geq 0$$

Στη συνέχεια θα παρουσιάσουμε τις πιο βασικές μετρικές για κάθε component ξεχωριστά :

Spout Metrics

$$\boldsymbol{Ack\ Rate} \quad \left(\frac{tuples}{sec}\right) = \frac{\sum_{i=1}^{n} acked_{e_i}(current) - \sum_{i=1}^{n} acked_{e_i}(last\ update)}{time(current) - time(last\ update)}$$

$$\boldsymbol{Emit\ Rate} \quad \left(\frac{tuples}{sec}\right) = \frac{\sum_{i=1}^{n} emitted_{e_i}(current) - \sum_{i=1}^{n} emitted_{e_i}(last\ update)}{time(current) - time(last\ update)}$$

$$\boldsymbol{Transfer\ Rate} \quad \left(\frac{tuples}{sec}\right) = \frac{\sum_{i=1}^{n} transfered_{e_i}(current) - \sum_{i=1}^{n} transfered_{e_i}(last\ update)}{time(current) - time(last\ update)}$$

$$\boldsymbol{Average\ Complete\ Latency} \quad (msec) = \frac{\sum_{i=1}^{n} complete\ latency_{e_i}(current) \cdot acked_{e_i}(current)}{\sum_{i=1}^{n} acked_{e_i}(current)}$$

Όπου e_i , $i = 1, .., n$ οι executors του component

Bolt Metrics

$$\boldsymbol{Ack\ Rate} \quad \left(\frac{tuples}{sec}\right)$$

$$\boldsymbol{Emit\ Rate} \quad \left(\frac{tuples}{sec}\right)$$

$$\boldsymbol{Transfer\ Rate} \quad \left(\frac{tuples}{sec}\right)$$

$$\boldsymbol{Execute\ Rate} \quad \left(\frac{tuples}{sec}\right) = \frac{\sum_{i=1}^{n} executed_{e_i}(current) - \sum_{i=1}^{n} executed_{e_i}(last\ update)}{time(current) - time(last\ update)}$$

$$\boldsymbol{Average\ Execute\ Latency}\quad (msec) = \frac{\sum_{i=1}^{n} execute\ latency_{e_i}(current)\ \cdot\ acked_{e_i}(current)}{\sum_{i=1}^{n} acked_{e_i}(current)}$$

$$\boldsymbol{Average\ Process\ Latency}\quad (msec) = \frac{\sum_{i=1}^{n} process\ latency_{e_i}(current)\ \cdot\ acked_{e_i}(current)}{\sum_{i=1}^{n} acked_{e_i}(current)}$$

$$\boldsymbol{Average\ Capacity}\ = \frac{\sum_{i=1}^{n} capacity_{e_i}(current)}{n}$$

$$\boldsymbol{Max\ Capacity}\ = \max_{i=1,\ldots,n} capacity_{e_i}(current)$$

3.2 Automatic Scaling (Rule Based)

3.2.1 Στόχος

Ο στόχος του συστήματος είναι να πετυχαίνει το target throughput που του δίνει ο χρήστης για μια τοπολογία. Θα πρέπει να παρακολουθεί συνεχώς το σύστημα καθώς τρέχει η τοπολογία και να αποφασίζει ποια/ποιες παραμέτρους παραλληλίας θα μεταβάλει για την τοπολογία και τα components της όταν αυτό κρίνεται απαραίτητο. Όσο δεν μπορεί να πετύχει τον στόχο θα πρέπει να συνεχίζει να ψάχνει για πιθανή επίλυση. Επίσης, αν φτάσει τον επιθυμητό στόχο θα πρέπει να συνεχίζει να παρακολουθεί το σύστημα σε περίπτωση που ξεφύγει εκ νέου από τον στόχο.

3.2.2 Βασική λογική

Ο μηχανισμός που θα υλοποιήσουμε πρέπει να πετυχαίνει ένα στόχο throughput. Για να επιτευχθεί αυτό θα πρέπει να βρει το βέλτιστο configuration για την τοπολογία.

Η τοπολογία θα πρέπει να ξεκινάει πάντα με μοναδιαίες όλες τις παραμέτρους → workers = 1 | executors for every component = 1. Το παραπάνω configuration είναι η ελάχιστη παραλληλία που μπορεί να δοθεί και θα χρησιμοποιείται σαν βάση για να ξεκινάει ο μηχανισμός.

Πως θα ξέρουμε όμως πως θα κινηθούμε για να βελτιώσουμε το throughput?
Ο σημαντικότερος παράγοντας που μπορεί να εμποδίζει το σύστημα από το να φτάσει σε υψηλό throughput είναι τα bottlenecks. Αυτά τα bottlenecks πρέπει να εντοπιστούν και να επιλυθούν.

Bottleneck στην τοπολογία θεωρείται ένα component το οποίο δεν μπορεί να φτάσει το ρυθμό επεξεργασίας που απαιτείται, δηλαδή να επεξεργαστεί τα δεδομένα με το ρυθμό που έρχονται από το προηγούμενο στάδιο. Αυτό συμβαίνει γιατί χρησιμοποιεί όλους τους διαθέσιμους πόρους που έχει και η λύση είναι να αυξηθεί η παραλληλία του component.

Σχήμα 3.2 - Bottleneck

Όπως δηλώνει και το όνομα του μηχανισμού, η λογική στηρίζεται σε βασικούς κανόνες για να αποφασίσει με ποιον τρόπο πρέπει να μεταβληθεί η τοπολογία.

Καθώς στόχος είναι η αύξηση του throughput της τοπολογίας το πρώτο πράγμα που πρέπει να επιλυθεί είναι τα πιθανά bottlenecks που υπάρχουν στην τοπολογία. Αυτά μπορεί να οφείλονται σε components που πιέζονται περισσότερο και δεν μπορούν να φτάσουν στις επιθυμητές επιδόσεις (η εύρεση του bottleneck μπορεί να γίνει από το capacity του κάθε component) ή στην συχνότερη περίπτωση σε μη επαρκής πόρους τοπολογίας.

Οι βασική λογική που ακολουθείται είναι :

- Αν component capacity $\gg$ τότε αυξάνουμε τα executors του component.
- Αν υπάρχουν πολλοί executors ανά worker τότε αυξάνουμε τα workers της τοπολογίας.
- Αν δεν υπάρχει κάποιο εμφανές bottleneck σε component και το άθροισμα των component capacity $\gg$ τότε αυξάνουμε τα workers της τοπολογίας
- Αν δεν υπάρχει κάποιο εμφανές bottleneck και το άθροισμα των component capacity $\ll$ τότε μειώνουμε τα workers της τοπολογίας. Σε αυτή την περίπτωση το spout έχει πιθανόν το πρόβλημα και δεν παρέχει δεδομένα με επαρκή ρυθμό άρα μειώνουμε resources.

Για τα παραπάνω ακολουθούμε τους παρακάτω κανόνες :

- capacity > 0.8 θεωρείται υψηλό
- sum capacities < 0.6 θεωρείται χαμηλό
- executors per worker $\leq$ cores of machine

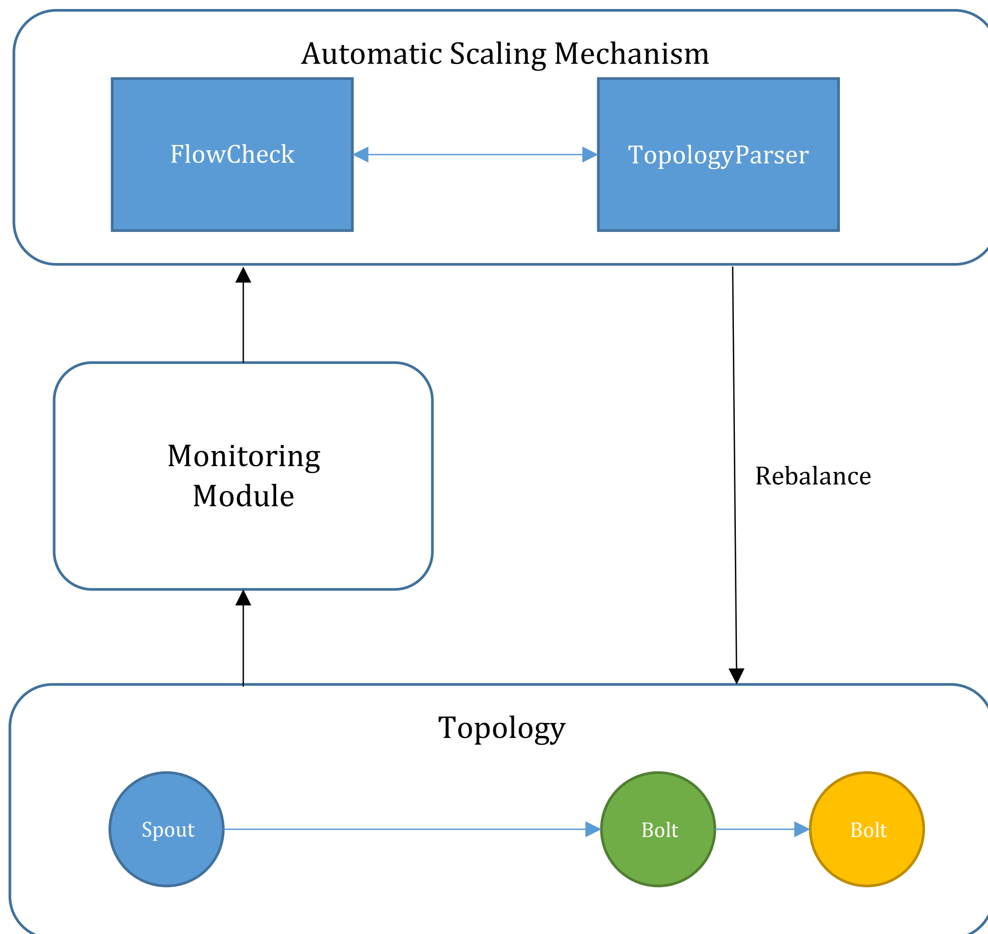
Τέλος, μετά από κάθε αλλαγή που κάνουμε στην τοπολογία και πριν προχωρήσουμε σε επόμενη αλλαγή αφήνουμε κάποιο χρονικό διάστημα για να σταθεροποιηθεί η επίδοση της τοπολογίας (warm up). Σε αυτό το διάστημα παρατηρούμε το throughput. Όσο βλέπουμε σχετική αύξησή του συνεχίζουμε να αφήνουμε την τοπολογία να τρέχει ως έχει. Μόλις δούμε ότι σταθεροποιείται (δηλαδή σχετική αύξηση/μείωση πολύ μικρή) ή ότι έχουμε μείωση της επίδοσης προχωράμε σε αλλαγή της τοπολογίας.
Η αλλαγή μπορεί και να μην γίνει κατευθείαν αλλά να συνεχίσουμε να παρακολουθούμε την τοπολογία ακόμη και αν είδαμε σχετική μείωση της επίδοσης (για να καλύψουμε την περίπτωση που απλώς έχουμε σφάλμα στα στατιστικά του Storm). Σε αυτή την περίπτωση αυξάνουμε ένα δείκτη severity και μόλις φτάσει σε ένα συγκεκριμένο όριο τότε προχωράμε σε αλλαγή.

Η σχετική αύξηση/μείωση του throughput ορίζεται ως:

$$\boldsymbol{relative\ throughput\ change} = \frac{throughput(current) - throughput(previous\ update)}{throughput(previous\ update)}$$

3.3 Αρχιτεκτονική

Η αρχιτεκτονική του μηχανισμού είναι δομημένη ως ένα σύστημα που συνδέεται τόσο με το monitoring μηχανισμό όσο και απευθείας με την ίδια την τοπολογία που τρέχει στο Storm cluster. Τρέχει αέναα, παρακολουθώντας την συμπεριφορά της τοπολογίας(μέσω των μετρικών) ανά συγκεκριμένα χρονικά διαστήματα και όποτε κρίνεται απαραίτητο παρεμβαίνει αλλάζοντας τις παραμέτρους παραλληλίας(workers/executors).



Σχήμα 3.3 – Architecture

Το σύστημα περιλαμβάνει δύο κύρια modules που συνδυάζονται για την λειτουργία του μηχανισμού. Αυτά είναι ο **TopologyParser** και ο **FlowChecker** για τα οποία θα δώσουμε μια σύντομη περιγραφή της λογικής τους και τον αλγόριθμο που ακολουθεί το καθένα σε αφαιρετικό επίπεδο.

Ο **TopologyParser** είναι υπεύθυνος για τις high-level λειτουργίες του μηχανισμού. Συγκεκριμένα αρχικοποιεί τη σύνδεση με το Monitoring διαβάζει τα configurations του χρήστη(στα οποία συμπεριλαμβάνεται η τοπολογία και ο στόχος του throughput σε tuples/sec) και τα μετατρέπει στη μορφή που απαιτείται για να ξεκινήσει ο μηχανισμός. Αφού εκτελέσει την αρχικοποίηση των δεδομένων παραμένει για πάντα σε loop το οποίο ανανεώνει τις μετρικές της τοπολογίας και καλεί το FlowCheck.

Από την άλλη το **FlowCheck** εκτελεί τις low-level λειτουργίες του μηχανισμού. Περιέχει τη λογική με την οποία ελέγχεται η επίτευξη του στόχου που δόθηκε από τον χρήστη και αποφασίζει για πιθανές βελτιώσεις στην τοπολογία αλλάζοντας τις παραμέτρους παραλληλίας. Χρησιμοποιεί τις μετρικές για να βγάλει

συμπεράσματα για την κατάσταση της τοπολογίας και έπειτα από έλεγχο προχωράει σε αλλαγές, αν είναι απαραίτητες. Όλα τα απαραίτητα δεδομένα για τη λειτουργία του τα λαμβάνει από τον TopologyParser.

Σαν γενική ιδέα, ο παραπάνω μηχανισμός αυξάνει σταδιακά τις παραμέτρους παραλληλίας, προσθέτοντας είτε executors σε components είτε workers με σκοπό να επιλύσει τα πιθανά bottlenecks που εμφανίζονται στην τοπολογία. Αυτό ακολουθείται όσο βλέπει ότι η τοπολογία δεν πετυχαίνει τον στόχο και παράλληλά χρησιμοποιεί όλους τους διαθέσιμους πόρους, πράγμα που σημαίνει ότι έχει περιθώρια αύξησης του throughput. Σε αντίθετη περίπτωση(δηλαδή δεν αξιοποιούνται όλοι οι πόροι του συστήματος) μειώνει τους πόρους.

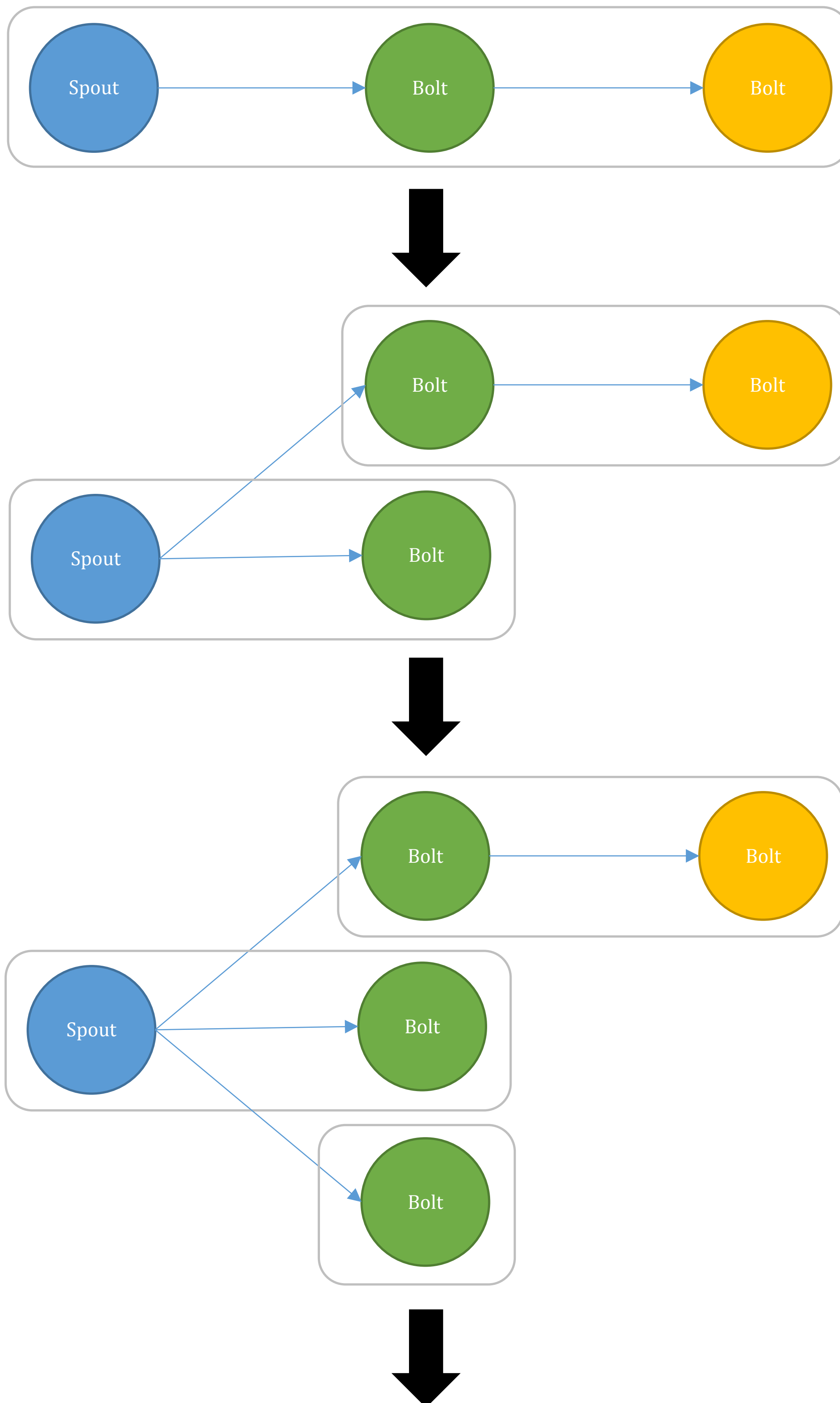
Με τον όρο σταδιακή αύξηση των παραμέτρων παραλληλίας εννοείται ότι για κάθε component στο οποίο εμφανίζεται bottleneck οι executors αυξάνονται κατά 1 τη φορά. Αυτό βέβαια μπορεί να γίνει για πολλά components που εμφανίζουν υψηλό capacity κατά την παρατήρηση. Επίσης στο ίδιο βήμα υπάρχει περίπτωση να αυξηθούν και οι workers της τοπολογίας αν το συνολικό άθροισμα των executors είναι μεγάλο, αλλά και πάλι κατά 1 τη φορά. (άρα παρόλο που γενικότερα ακολουθείται σταδιακή αύξηση, υπάρχει περίπτωση να γίνει και ένα σύνολο αλλαγών στην τοπολογία σε ένα βήμα).

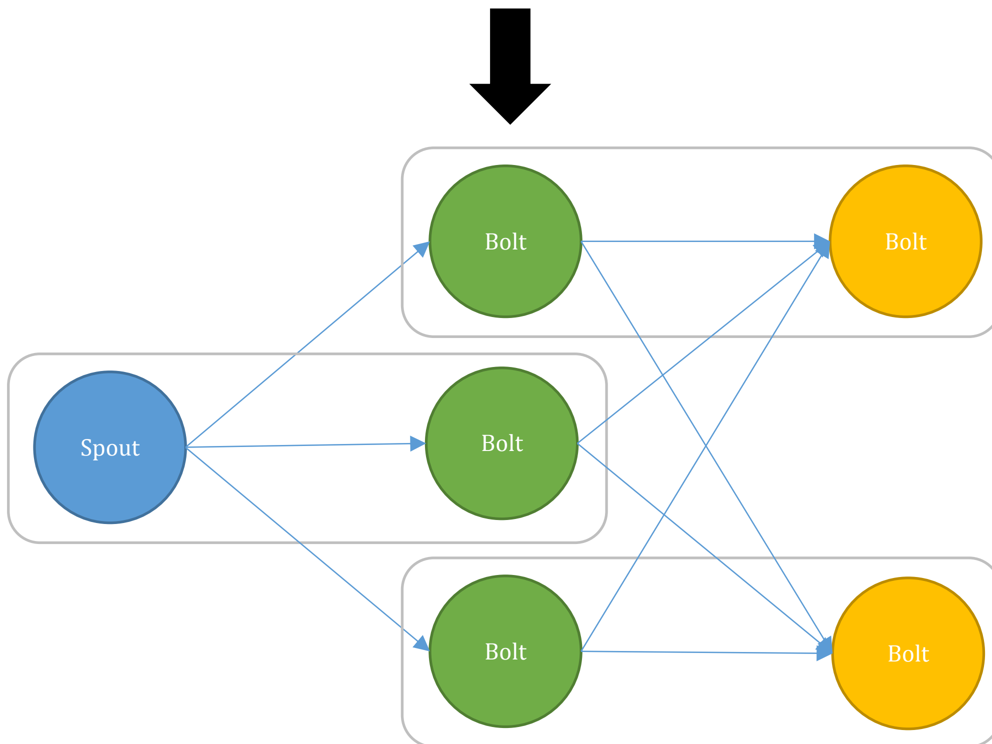
TopologyParser

```
1.     read topology file
2.     create topology graph
3.     read target throughput
4.     initialize metrics
5.     flow ← initialize FlowCheck module
6.     while true do
7.         wait
8.         update metrics
9.         f.startCheck
10.        if f.isRebalanced do
11.            wait
12.            initialize metrics
13.            f.refresh
14.        end if
15.    end while
```

```
FlowCheck

1.   root 🡐 spout components
2.   severity 🡐 0
3.   target 🡐 target throughput (for every root)
4.   config 🡐 new configuration
5.
6.   for every root component do
7.       if root.throughput < target(root) do
8.           change 🡐 relative throughput(current, previous)
9.           if change > 0 && change < 0.05 do
10.              severity++
11.          else if change > 0 && change > 0.05 do
12.              severity--
13.          else if change < 0 && change < -0.02 do
14.              severity++
15.          end if
16.      end if
17.      if severity > 2 do
18.          root.needsCheck
19.      end if
20.  end for
21.
22.  for every root who needsCheck do
23.      pq 🡐 bolt components in priority queue(by max capacity)
24.      bolt 🡐 pq.poll
25.      while bolt.maxCapacity > 0.8 do
26.          if !config.contains(bolt) do
27.              config.add(bolt, executors++)
28.          end if
29.      end while
30.  end for
31.
32.  if config.isEmpty do
33.      if totalCapacity(sum capacities) < 0.6 do
34.          config.workers--
35.      else
36.          config.workers++
37.      end if
38.  else
39.      if totalExecutors(sum executors) > cores*workers do
40.          config.workers++
41.      end if
42.  end if
43.  commitRebalance(config)
```

Παρακάτω φαίνεται σχηματικά ο τρόπος με τον οποίο λειτουργεί ο μηχανισμός.





Σχήμα 3.4 - Rule Based Topology Changes

3.4 Configuration Finder (Machine Learning)

3.4.1 Στόχος

Ο στόχος του μηχανισμού είναι η εύρεση του βέλτιστου/βέλτιστων configuration, είτε δεδομένων time/money constraints που δίνονται από τον χρήστη, είτε χωρίς constraints. Αυτό θα πρέπει να γίνεται κατευθείαν και χωρίς ο μηχανισμός να χρειάζεται να παρακολουθήσει την τοπολογία ολόκληρη να τρέχει στο Storm cluster. Για να επιτευχθεί ο παραπάνω στόχος χρησιμοποιείται machine learning με τη μορφή μοντέλων. Έτσι για κάθε πιθανό configuration μπορεί να προβλεφθεί η επίδοση που προσφέρει και τελικά να επιλεχθεί το βέλτιστο.

3.4.2 Βασική λογική

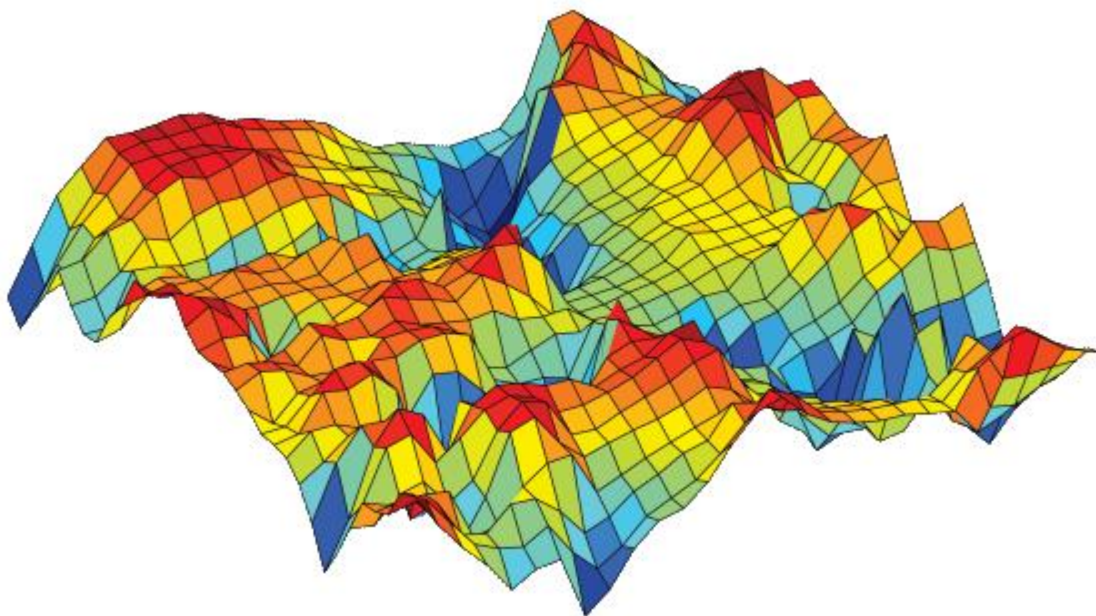
Η βάση όλου του μηχανισμού είναι τα μοντέλα των components της τοπολογίας. Για κάθε component της τοπολογίας χρησιμοποιείται ένα μοντέλο, από το οποίο εξάγονται προβλέψεις για τη συμπεριφορά του στη συνολική τοπολογία ανάλογα με το configuration της.

Η ιδέα για τον μηχανισμό είναι ότι θα πρέπει να είναι επεκτάσιμος. Γι' αυτό το λόγο επιλέχθηκε τα μοντέλα να δημιουργούνται σε επίπεδο component και όχι τοπολογίας. Πολλές τοπολογίες μπορεί να χρησιμοποιούν παρόμοια ή ίδια components. Έτσι τα μοντέλα που έχουν δημιουργηθεί για τα components μιας τοπολογίας μπορούν να επαναχρησιμοποιηθούν(και να εμπλουτιστούν) και σε πολλές άλλες. Με αυτόν τον τρόπο δημιουργούνται βιβλιοθήκες από μοντέλα components που μπορούν πρώτον να επαναχρησιμοποιηθούν και δεύτερον να επεκταθούν, βελτιωθούν.

Για να δημιουργηθεί το μοντέλο ενός component περνάει από stress tests. Όλα τα stress tests εκτελούνται σε τοπολογίες 2 component, το ένα είναι το subject bolt component το οποίο θέλουμε να μοντελοποιήσουμε και το άλλο είναι ένα spout component που παράγει δεδομένα και τα στέλνει στο bolt. Για να μπορέσουμε να κατασκευάσουμε ένα ολοκληρωμένο μοντέλο τρέχουμε την παραπάνω τοπολογία με διαφορετικά configuration και παίρνουμε από το monitoring τις αντίστοιχες μετρικές που χρειαζόμαστε.

Αφού παραχθούν όλα τα απαραίτητα μοντέλα για τα components, χρησιμοποιούνται για να υπολογιστεί το throughput της συνολικής τοπολογίας για ένα πιθανό configuration. Ο προβλέπτης δίνει μια πρόβλεψη για καθένα από τα components της τοπολογίας και όλες μαζί συνδυάζονται από τον throughput calculator για να παραχθεί το τελικό throughput.

Για να βρεθεί το βέλτιστο configuration, δηλαδή αυτό που δίνει το μέγιστο throughput θα πρέπει να γίνει αναζήτηση στον χώρο των πιθανών λύσεων(configuration). Αυτό γίνεται με ευριστικούς αλγόριθμους που χρησιμοποιούν ως «δυναμικό» το throughput που παράγεται-προβλέπεται από τον throughput calculator.



Σχήμα 3.5 - Search Space

Όσο μεγαλώνει η τοπολογία τόσο αυξάνονται οι μεταβλητές που πρέπει να λάβουμε υπόψιν. Αυτό σημαίνει ότι προστίθενται διαστάσεις στο search space και το πρόβλημα γίνεται πιο πολύπλοκο.

3.4.3 Αρχιτεκτονική

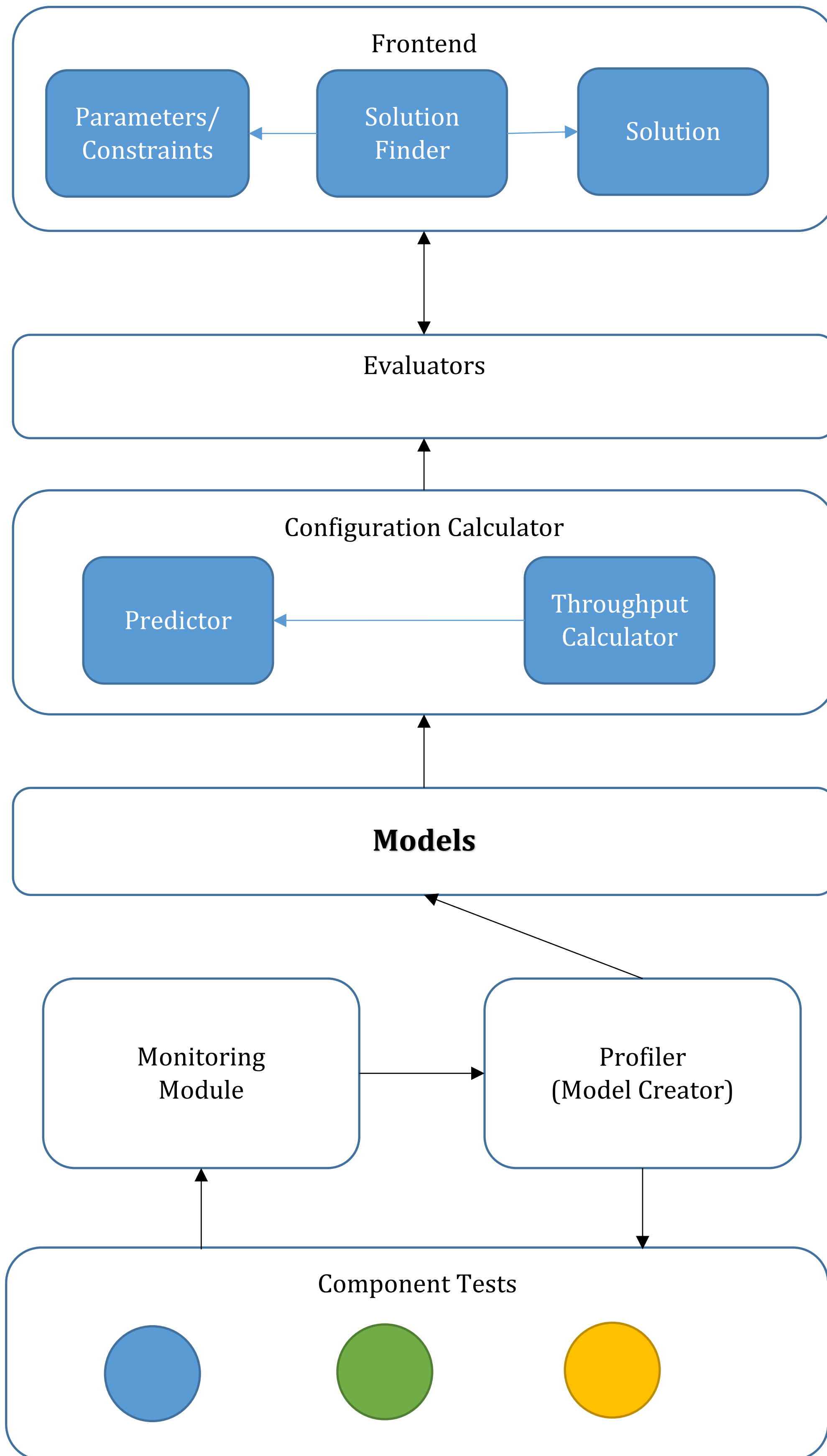
Ήδη από όσα είπαμε προηγουμένως φαίνεται ο τρόπος που δομείται ο μηχανισμός. Θα παρουσιάσουμε το σύνολο της αρχιτεκτονικής του μηχανισμού σχηματικά και θα αναλύσουμε τα επιμέρους κομμάτια του.

Profiler

Είναι το κομμάτι του μηχανισμού που αναλαμβάνει να εκτελέσει τα stress tests για τα components, να συγκεντρώσει τις μετρικές από το monitoring και να δημιουργήσει το αντίστοιχο μοντέλο για το κάθε component.
Συγκεκριμένα εκτελεί nested loops καθένα από τα οποία παριστάνει μια μεταβλητή παράμετρο παραλληλίας. Μέσα σε κάθε εσωτερικό loop δημιουργεί μια τοπολογία, την τρέχει στο Storm cluster, παρακολουθεί την πορεία των μετρήσεων και τέλος τερματίζει την τοπολογία.

```
Profiler

1.     for workers do
2.         for spout executors do
3.             for bolt executors do
4.                 buildTopology(#spouts, #bolts)
5.                 submitTopologyToCluster(#workers)
6.                 monitorTopology
7.                 killTopology
8.                 wait
9.             end for
10.        end for
11.    end for
```

Όταν τελειώσει το monitoring για ένα configuration της τοπολογίας, οι μετρικές που συλλέχθηκαν γίνονται append σε ένα file για το component. Μόλις ολοκληρωθούν όλα τα configuration tests θα έχουμε ένα file που περιγράφει τη συμπεριφορά του component. Αυτό το file αποτελεί τη βάση για τη δημιουργία του μοντέλου του component (ARFF file).



Σχήμα 3.6 - Configuration Finder Architecture

Models

Τα μοντέλα είναι σε επίπεδο component. Για τη δημιουργία της χρησιμοποιούμε το εργαλείο WEKA [9], το οποίο χρησιμοποιεί τα files που δημιουργήσαμε με τον profiler. Συγκεκριμένα παίρνει τις μετρήσεις του file ως dataset και δημιουργεί ένα μοντέλο για το component βάσει αυτού ή μιας φιλτραρισμένης εκδοχής του. Στη συνέχεια το μοντέλο μπορεί να χρησιμοποιηθεί για να προβλεφθεί η συμπεριφορά του component για πιθανά configurations.

Κάθε component ARFF file περιέχει τα εξής attributes:

1. workers
2. boltExecutors
3. spoutExecutors
4. totalLatency
5. acked/sec
6. executeLatency
7. avgCapacity
8. emitted/sec

Τα μοντέλο που παράγεται από το WEKA δεν είναι απαραίτητο ότι λαμβάνει υπόψιν του όλα τα παραπάνω attributes (μάλιστα είναι προτιμότερο να δουλεύουμε με περισσότερα μοντέλα για ένα component που χρησιμοποιούν υποσύνολο των παραπάνω attributes για να έχουμε καλύτερες προβλέψεις)

Predictor

Ο προβλέπτης είναι αυτός που χρησιμοποιεί τα μοντέλα για να προβλέψει την επίδοση των components. Ο predictor και το αντίστοιχο μοντέλο που χρησιμοποιεί είναι δεμένα μαζί, με την έννοια ότι το μοντέλο χτίζεται την στιγμή που ζητείται να γίνει μια πρόβλεψη. Συγκεκριμένα φιλτράρεται το dataset, το μοντέλο εκπαιδεύεται(train classifier) βάσει ενός classifier που έχει προεπιλεχθεί και τέλος δημιουργείται η πρόβλεψη.

```
Predictor

1.    classifier ← algorithm of choice
2.    dataset ← load from ARFF file
3.    filtered ← filter(dataset)
4.    classifier.train(filtered)
5.    prediction ← createPrediction(configuration)
6.    classifier.classify(prediction)
```

Για παράδειγμα αν θέλουμε να προβλέψουμε το execute latency ενός bolt με configuration (2 workers, 2 spout executors, 2 bolt executors) θα εκτελεστούν τα παρακάτω :

- Φιλτράρεται το dataset και κρατάμε μόνο τα attributes 1, 2, 3, 6
- Γίνεται train ο classifier(π.χ. MultilayerPerceptron)
- Δημιουργείται ένα prediction με τη μορφή (2, 2, 2, ?)
- παίρνουμε την πρόβλεψη για το execute latency από τον classifier (?=classifier output)

Ανάλογα με το επιθυμητό prediction γίνεται το αντίστοιχο φιλτράρισμα.

Throughput Calculator

Το module αυτό είναι υπεύθυνο για τον υπολογισμό του throughput μιας τοπολογίας. Δέχεται ως input την τοπολογία και το configuration(workers, executors) της και επιστρέφει το throughput. Για να το πετύχει αυτό χρησιμοποιεί τον predictor(ή πιο σωστά πολλαπλά instances του predictor, όσα είναι και τα διαφορετικά components της τοπολογίας). Το σημαντικότερο κομμάτι, όμως, είναι ο αλγόριθμος με βάση τον οποίο συνδυάζονται οι προβλέψεις για να προκύψει το τελικό throughput.

Σε αυτό το σημείο, πριν καταλήξουμε στον τελικό αλγόριθμο που χρησιμοποιούμε, είναι σημαντικό να κατανοηθεί το πρόβλημα που πρέπει να επιλύσουμε. Παρακάτω θα παρουσιάσουμε σταδιακά τη λογική με την οποία δομείται ο αλγόριθμος.

Αρχικά θα δέσουμε όλες τις έννοιες που χρησιμοποιούμε ως τώρα για να μας δώσουν σαν αποτέλεσμα το throughput.

Το σύστημα έχει στη διάθεση του κάποιους πόρους οι οποίοι μπορούν να μεταφραστούν σε μονάδες χρόνου(time slices). Κάθε worker αντιπροσωπεύει μια μονάδα χρόνου η οποία αντιστοιχεί σε ένα 1 sec. Έτσι αν έχουμε 4 workers έχουμε στη διάθεση μας 4 μονάδες χρόνου, δηλαδή 4 sec.

Κάθε component έχει συγκεκριμένο αριθμό executors. Αυτό σημαίνει ότι μπορεί να εκμεταλλευτεί από τους παραπάνω πόρους ένα περιορισμένο μέγεθος. Αν για παράδειγμα έχουμε στο σύστημα μας 4 sec, αλλά το component έχει 2 executors, τότε μπορεί να εκμεταλλευτεί μόνο 2 sec.

Από τα 2 παραπάνω έχουμε :

$$time(component) = \min(workers, executors)$$

Κάθε component χαρακτηρίζεται από το execute latency του. Για να συνδυάσουμε όλα τα παραπάνω, η συσχέτιση μεταξύ throughput – execute latency – time έχει ως εξής :

$$throughput(component) = \frac{time}{execute\ latency} \quad (tuples/sec)$$

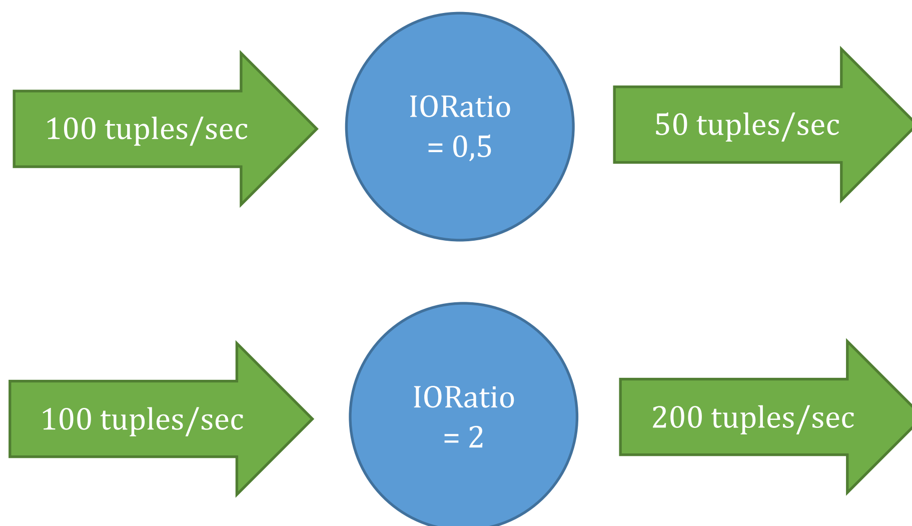
Για παράδειγμα αν ένα component έχει 1 executor το throughput του είναι :

$$throughput = \frac{1}{execute\ latency}$$

Το παραπάνω παράδειγμα δέχεται ως δεδομένο ότι ο executor του component έχει στη διάθεση του ολόκληρη τη μονάδα χρόνου. Αυτό δεν συμβαίνει πάντα καθώς μπορεί να συνυπάρχει με άλλα executors στον ίδιο worker και να μοιράζεται τους πόρους.

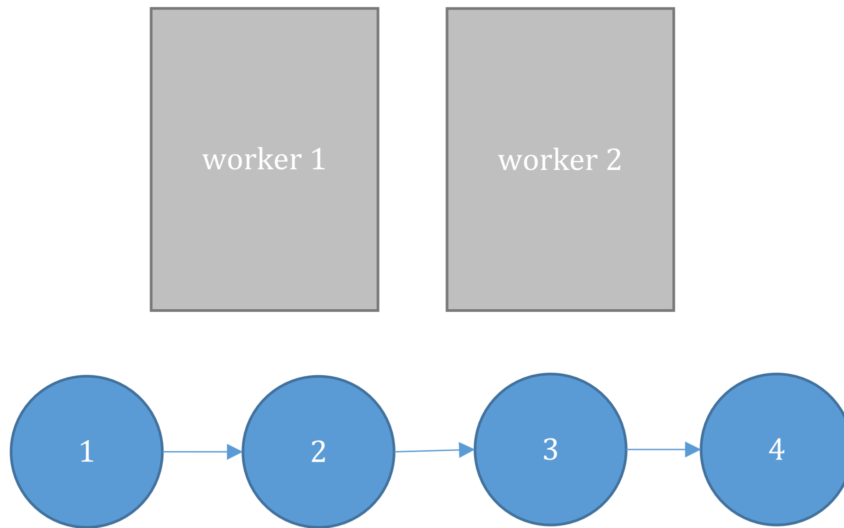
Χρήσιμος δείκτης για ένα component είναι το IORatio που δείχνει την διαφορά μεταξύ των εισερχόμενων και εξερχόμενων δεδομένων.

$$IORatio = \frac{executed}{emitted}$$



Σε αυτό το σημείο θα παρουσιάσουμε το πρόβλημα που πρέπει να λύσουμε και στη συνέχεια θα παρουσιάσουμε τις πιθανές λύσεις. Επειδή περιλαμβάνει πολλές μεταβλητές θα το παρουσιάσουμε με τη βοήθεια παραδείγματος.

Έχουμε μια τοπολογία με 4 components και 2 workers :



$$execute\ latency = el, \qquad IORatio = io$$

$$el_1 = el_3 = 1ms\ , el_2 = el_4 = 2ms$$

$$io_1 = io_2 = io_3 = io_4 = 1$$

$$\forall component, executors = 1$$

$$\forall\ worker, add\ time\ slice \xRightarrow{workers=2} time = 2$$

έστω ότι $throughput = x$, τότε για βρούμε το μέγιστο throughput πρέπει να βρούμε το x για το οποίο ισχύουν τα παρακάτω :

1. $x \cdot el_1 \leq \min(executors, workers) = 1$
2. $x \cdot el_2 \leq \min(executors, workers) = 1$
3. $x \cdot el_3 \leq \min(executors, workers) = 1$
4. $x \cdot el_4 \leq \min(executors, workers) = 1$
5. $x \cdot el_1 + x \cdot el_2 + x \cdot el_3 + x \cdot el_4 \leq \min(sum(executors), workers) = 2$

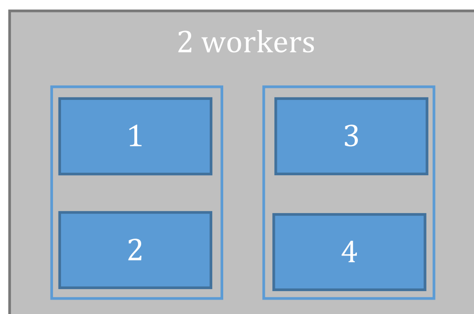
$$\max throughput = x = \min(1,2,3,4,5)$$

Με μια πρώτη ματιά μπορεί κανείς να πει ότι είναι εύκολο να βρει το μέγιστο throughput και ότι είναι :

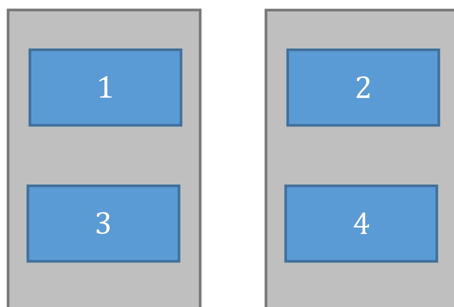
$$\max throughput = x = \frac{2}{el_1 + el_2 + el_3 + el_4}$$

Αυτό όμως ενώ είναι σωστό σαν σκέψη κρύβει ένα misconception. Συγκεκριμένα από τις παραπάνω ανισότητες(1-4) μπορεί να καταλάβει κανείς ότι κάθε component δεν ενδιαφέρεται για το που βρίσκεται ή για το αν έχει τοποθετηθεί μαζί με άλλα στον ίδιο worker. Έτσι βλέπει ότι έχει όλη τη μονάδα χρόνου στη διάθεση του. Είναι δηλαδή σαν να βλέπει virtual time όπως αντίστοιχα οι εφαρμογές βλέπουν virtual RAM και νομίζουν ότι είναι όλη στη διάθεση τους αλλά στην πραγματικότητα δεν είναι. Αυτό έχει ως αποτέλεσμα να οδηγηθούμε σε υπερεκτίμηση του throughput. Η τελευταία ανισότητα(5) περιορίζει την πιθανή υπερεκτίμηση, αλλά λόγω του αθροίσματος χάνουμε τη διακριτότητα (όλοι οι διακριτοί workers που προσφέρουν από μια μονάδα χρόνου αθροίζονται σε ένα μεγάλο pool με συνολικό χρόνο) και κατά συνέπεια πιθανούς περιορισμούς που μπορεί να προκύψουν από την τοποθέτηση των executors στους workers.

Ο καλύτερος τρόπος για να δείξουμε το παραπάνω είναι με ένα αντιπαράδειγμα. Ας θεωρήσουμε το ίδιο πρόβλημα με ίδιο αριθμό workers και ίδια τοπολογία.



$$throughput = \frac{2\,s}{(1+2+1+2)ms} = 333\ tuples/sec$$



$$throughput = \min\left(\frac{1\,s}{(1+1)ms}, \frac{1\,s}{(2+2)ms}\right) = 250\ tuples/sec$$

Τα δύο παραπάνω είναι δύο διαφορετικές προσεγγίσεις για το πως υπολογίζεται το μέγιστο throughput μιας τοπολογίας και αντιπροσωπεύουν δύο διαφορετικούς αλγόριθμους που μπορούν να χρησιμοποιηθούν.

1. Η πρώτη προσέγγιση είναι Schedule-Agnostic με την έννοια ότι δεν ξέρει με ποιον τρόπο έχουν τοποθετηθεί οι executors στους workers. Θεωρεί ότι ο scheduler κανονίζει για την βέλτιστη τοποθέτηση των executors στους workers. Έτσι είναι σαν να βλέπει αντί για διακριτούς workers ένα time pool. Μειονέκτημα αυτής της προσέγγισης είναι ότι σε συγκεκριμένες

περιπτώσεις μπορεί να υπερεκτιμήσει το throughput. Τέλος θα πρέπει να ελεγχθούν όλες οι ανισότητες 1-5 για να προκύψει το σωστό αποτέλεσμα.

2. Η δεύτερη προσέγγιση λαμβάνει υπόψιν της τον Scheduler(για το Storm ο default αλγόριθμός scheduling είναι round-robin). Γνωρίζει, λοιπόν, την τοποθέτηση των executors στους workers και κατά συνέπεια μπορεί να υπολογίσει με μεγάλη ακρίβεια το throughput.

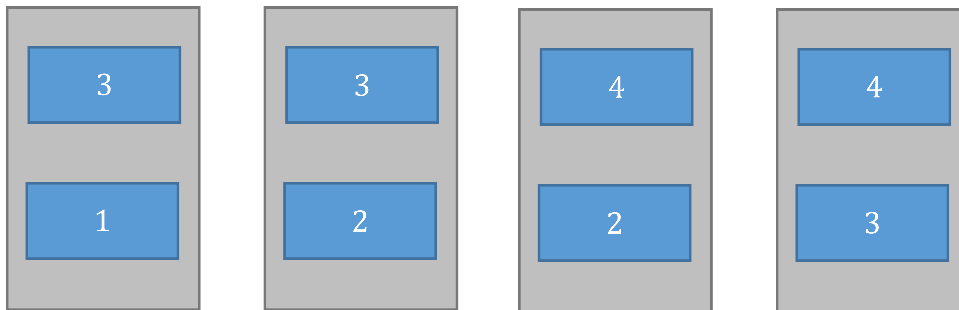
Για την δεύτερη προσέγγιση θα πρέπει να δώσουμε καινούριες ανισότητες για να καλύψουμε τους περιορισμούς που προκύπτουν ανά worker:

1. $x \cdot el_1 + x \cdot el_3 \leq 1$
2. $x \cdot el_2 + x \cdot el_4 \leq 1$

Η επίλυση του παραπάνω μπορεί να φαίνεται απλή, αλλά η πολυπλοκότητα αυξάνεται εκθετικά αν οι μεταβλητές παραλληλίας αλλάξουν. Ας κρατήσουμε την ίδια τοπολογία με τα 4 components και να δούμε την εξής περίπτωση:

$$workers = 4$$

$$executors_1 = 1, executors_2 = 2, executors_3 = 3, executors_4 = 2$$



$$output_{component,executor} = o_{i,j}$$

1. $o_{1,1} = x$
2. $o_{2,1} + o_{2,2} = x$
3. $o_{3,1} + o_{3,2} + o_{3,3} = x$
4. $o_{4,1} + o_{4,2} = x$
5. $o_{1,1} \cdot el_1 + o_{3,2} \cdot el_3 \leq 1$
6. $o_{2,1} \cdot el_2 + o_{3,3} \cdot el_3 \leq 1$
7. $o_{2,2} \cdot el_2 + o_{4,1} \cdot el_4 \leq 1$
8. $o_{3,1} \cdot el_3 + o_{4,2} \cdot el_4 \leq 1$

Επειδή τα παραπάνω προσθέτουν μεγάλη πολυπλοκότητα αυτό που θα κάνουμε είναι αντί να λύσουμε το παραπάνω σύστημα ανισοτήτων, θα δημιουργήσουμε μια μοντελοποίηση του συστήματος που να ικανοποιεί όλους τους παραπάνω περιορισμούς.

Η βασική λογική ακολουθεί τα παρακάτω βήματα:

- Δημιουργούνται containers που αντιστοιχούν στους workers. Ο αριθμός τους είναι όσοι και οι workers του configuration. Κάθε container έχει ένα χαρακτηρίζεται από ένα watermark που δείχνει το ποσοστό χρήσης του. Επίσης ορίζεται το capacity του container που είναι το μέγιστο watermark.
- Οι operators που αντιστοιχούν στα components της τοπολογίας γίνονται assign με round-robin τρόπο στους containers. Κάθε operator κρατάει μια priority queue με τους containers που έχει γίνει assign, η ταξινόμηση είναι κατά container watermark.
- Με τα δύο παραπάνω βήματα έχουμε την δομή της τοπολογίας στο cluster και μένει να βρούμε το μέγιστο throughput που μπορούμε να πετύχουμε με το συγκεκριμένο configuration. Αυτό γίνεται με σταδιακό γέμισμα των containers από τα διαφορετικά operators. Σε κάθε βήμα προσομοιάζουμε τον τρόπο με τον οποίο κυκλοφορεί και εκτελείται ένα tuple της τοπολογίας. Συγκεκριμένα, από το spout υπολογίζουμε τον ελάχιστο αριθμό tuples που μπορεί να παράγει και αυτό αποτελεί το βήμα μας. Κάθε tuple το ακολουθούμε από την δημιουργία του και για όλα τα στάδια εκτέλεσης αυξάνοντας σε κάθε βήμα το watermark στα αντίστοιχα containers. Έτσι, όταν σε κάποιο από τα containers το watermark φτάσει στο μέγιστο ξέρουμε ότι η τοπολογία δεν μπορεί να δεχτεί παραπάνω throughput και αυτό που έχουμε υπολογίσει μέχρι εκείνη τη στιγμή είναι το max. Να διευκρινίσουμε ότι δεν χρειάζεται να εκτελέσουμε την προσομοίωση της διαδρομής σε κάθε βήμα, αυτό γίνεται μια φορά στην αρχή και στα υπόλοιπα βήματα απλά προσθέτουμε τους επιπρόσθετους χρόνους στα κατάλληλα containers.

```
Throughout Calculator

1.    queue
2.    operatorList
3.    for every spout do
4.        assign ← assign operator to containers
5.        operatorlist.add(assign)
6.        for every neighbor of spout do
7.            queue.add(neighbor)
8.        end for
9.    end for
10.   while !queue.isEmpty do
11.       temp ← queue.poll
12.       assign ← assign operator to containers
13.       operatorlist.add(assign)
14.       for every neighbor of temp do
15.           queue.add(neighbor)
16.       end for
17.   end while
18.   while containers are not full do
19.       for every operator in operatorList do
20.           operator.executeStep
21.       end for
22.   end while
```

Evaluators

Οι evaluators είναι αυτοί που χτίζουν το high level constrained problem. Στον χρήστη έχουμε δώσει τη δυνατότητα να δίνει απαιτήσεις για money και time. Αυτό δημιουργεί 4 διαφορετικά προβλήματα που χρειάζονται επίλυση (ανάλογα με το input που θα δώσει). Αυτά είναι :

1. Money Constraint Problem
2. Time Constraint Problem
3. Money+Time Constraint Problem
4. No Constraint Problem (Non dominated Pareto solutions)

Κάθε evaluator υλοποιεί ένα πρόβλημα από τα παραπάνω. Αυτό σημαίνει ότι παρέχει τη δυνατότητα να παραγάγει ένα τυχαίο σημείο του search space που αντιστοιχεί σε ένα configuration της τοπολογίας, όπως επίσης να

πραγματοποιήσει και evaluation αυτού του σημείου. Για το evaluation χρησιμοποιείται ο throughput calculator για να υπολογίσουμε το «δυναμικό» του σημείου και τέλος περνάει και από έλεγχο αν ικανοποιεί τα constraints του προβλήματος.

Τα time-money constraints μπορούν να αναχθούν σε throughput ως εξής :

$$time = \frac{dataset\ size}{throughput}$$

$$money = workers \cdot cost\left(\frac{worker}{sec}\right) \cdot time$$

Front End

Το front end είναι αυτό που φαίνεται στη μεριά του χρήστη. Ο χρήστης θέτει τις παραμέτρους, οι συστημικές αφορούν το cluster και οι user είναι οι επιλογές του χρήστη(δηλαδή τα constraints). Αυτό που του επιστρέφεται είναι ένα σύνολο λύσεων. Η πληροφορία για την κάθε λύση είναι συγκεντρωμένη στο UserSolution.

Το module που δένει τα δύο προηγούμενα είναι ο SolutionFinder. Δέχεται ως είσοδο τις παραμέτρους του χρήστη και του επιστρέφει τις λύσεις. Για την εύρεση της λύσης δημιουργεί ένα από τα παραπάνω προβλήματα(ανάλογα με το τι έχει επιλέξει ο χρήστης) και εκτελεί αναζήτηση στον χώρο των λύσεων. Επιστρέφει την βέλτιστη/βέλτιστες λύσεις στο πρόβλημα που δημιουργήθηκε ή δεν επιστρέφει τίποτα αν δεν μπόρεσε να βρει λύση που να ικανοποιεί τις απαιτήσεις του χρήστη.

Για το χτίσιμο των προβλημάτων και την αναζήτηση στον χώρο των λύσεων χρησιμοποιήσαμε το MOEA Framework [10] που παρέχει τα κατάλληλα εργαλεία για να μοντελοποιήσουμε τα constrained problems και τους αλγόριθμους για να εκτελέσουμε το search.

4 Πειραματική Αποτίμηση

Σε αυτό το σημείο θα προχωρήσουμε σε πειραματική εκτίμηση των συστημάτων που υλοποιήσαμε. Θα εκτελέσουμε τόσο πειράματα επιδόσεων και αποτελεσματικότητας και για τους δύο μηχανισμούς και επιπρόσθετα για τον predictor θα εκτελέσουμε και πειράματα ακρίβειας.

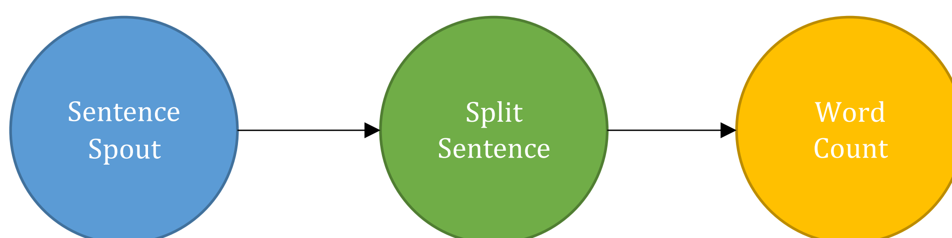
4.1 Πειραματικό Περιβάλλον

Το Apache Storm Cluster δημιουργήθηκε στο cloud περιβάλλον του Openstack. Στην διάθεση μας έχουμε 8 VMs από τα οποία 1 χρησιμοποιήθηκε ως master και 7 ως slaves. Τα μηχανήματα έχουν τα ίδια τεχνικά χαρακτηριστικά (2 cores / 4GB RAM). Για τα πειράματα που θα εκτελέσουμε θεωρούμε ότι τα μηχανήματα έχουν την ίδια δυναμική και ομοιόμορφη συμπεριφορά.

Για τα πειράματα θα χρησιμοποιήσουμε 4 τοπολογίες με διαφορετικά χαρακτηριστικά και δομή η κάθε μία. Σκοπός μας είναι να χρησιμοποιήσουμε τοπολογίες με διαφορετικές δομικές μονάδες αλλά και διαφορετικό flow ώστε να μπορέσουμε να δούμε πως συμπεριφέρονται οι μηχανισμοί που υλοποιήσαμε στα διαφορετικά σενάρια.

WordCount Topology

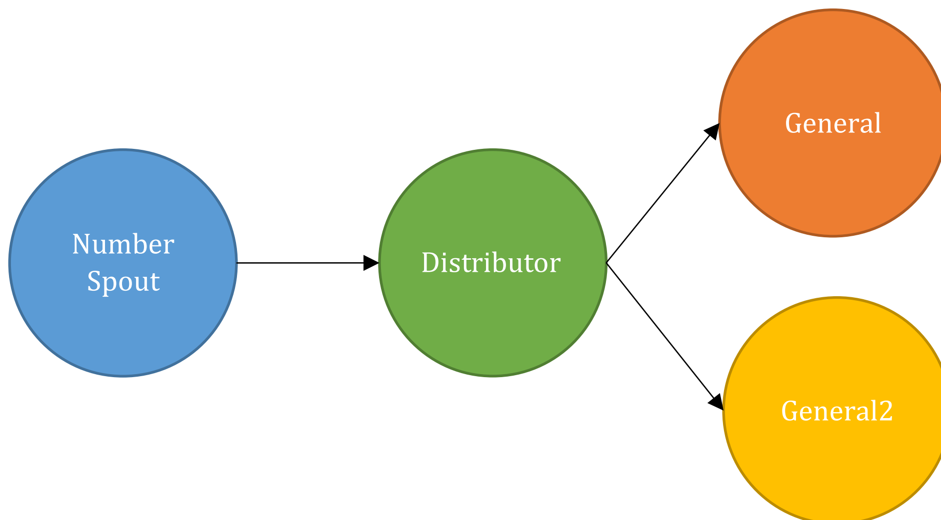
Η τοπολογία WordCount αποτελεί ένα από τα βασικά παραδείγματα που προσφέρονται για την κατανόηση των κατανεμημένων συστημάτων. Θα την χρησιμοποιήσουμε και στην περίπτωση μας αφού προσφέρει έναν συνδυασμό cpu intensive και network intensive διεργασίας.



Το Sentence Spout παράγει προτάσεις. Το Split Sentence δέχεται τις προτάσεις, τις χωρίζει με βάση τα κενά και στέλνει κάθε ξεχωριστεί λέξη. Το Word Count μετράει τις λέξεις που του έρχονται και βγάζει το συνολικό άθροισμα ανά λέξη.

CyberShake Topology

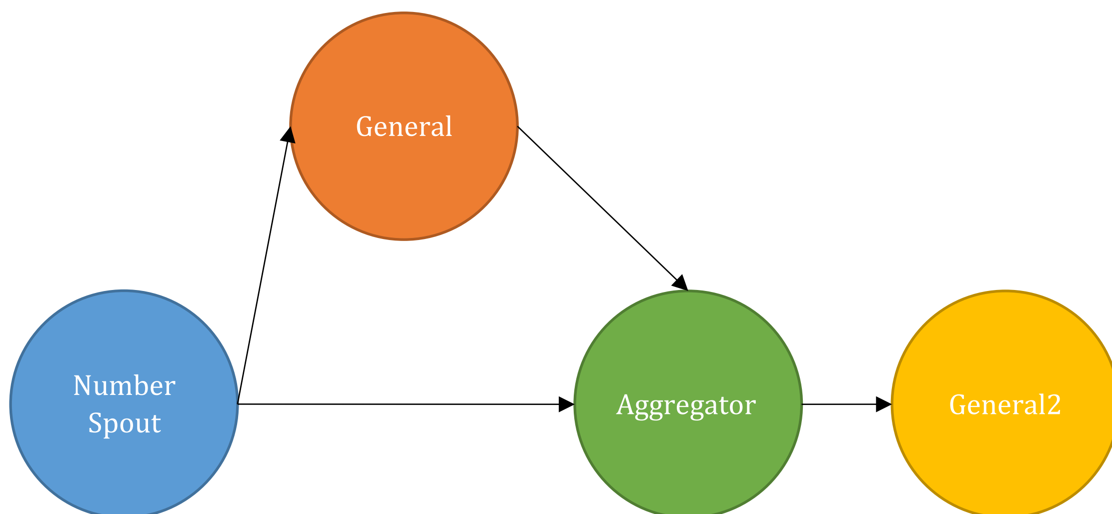
Η τοπολογία CyberShake στηρίζεται σε scientific workflow για να υπολογίζονται οι κίνδυνοι σεισμού. Τη φέραμε στα πλαίσια της εργασίας για να ταιριάζει με το μοντέλο του stream processing και αντιπροσωπεύει ένα καλό παράδειγμα network intensive διεργασίας.



Το Number Spout παράγει αριθμούς. Ο Distributor διαβάζει τα εισερχόμενα νούμερα και στέλνει ένα αντίγραφο σε καθένα από τα General, General2 τα οποία λειτουργούν σαν FIFO ουρές, δηλαδή διαβάζουν τον εισερχόμενο αριθμό και απλώς τον προωθούν στο επόμενο επίπεδο.

Montage Topology

Η τοπολογία Montage στηρίζεται σε scientific workflow που χρησιμοποιείται από τη NASA για τη συνένωση εικόνων. Τη φέραμε στις ανάγκες της μας για να συμπεριλάβουμε ένα πιο ιδιαίτερο flow. Αντιπροσωπεύει ένα καλό παράδειγμα network και memory intensive διεργασίας.

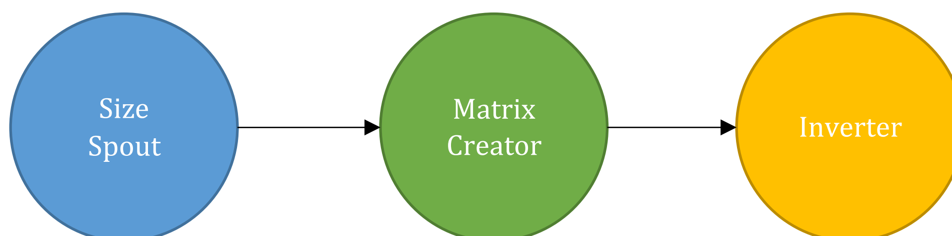


Το Number Spout παράγει δύο αριθμούς και τους στέλνει στον Aggregator και το General. Το General απλώς προωθεί τον αριθμό στον Aggregator. Ο Aggregator συνδυάζει ένα νούμερο από το Number Spout και ένα από το General και στέλνει

το αποτέλεσμα στο General2. Αν δεχθεί πολλαπλά μηνύματα από ένα component τα αποθηκεύει σε ένα queue μέχρι να έρθουν και από το δεύτερο.

Matrix Topology

Η τοπολογία Matrix είναι custom και μπορεί να επεκταθεί κατά βούληση για να εκτελεί οποιεσδήποτε πράξεις πινάκων. Το flow είναι απλό και αποτελεί ένα καλό παράδειγμα cpu και memory intensive διεργασίας.



Το Size Spout παράγει έναν αριθμό που αντιπροσωπεύει τις διαστάσεις ενός τετραγωνικού πίνακα. Το Matrix Creator δέχεται τον αριθμό n και παράγει έναν τυχαίο τετραγωνικό πίνακα με διαστάσεις $n \times n$. Ο Inverter δέχεται τον πίνακα, υπολογίζει τον αντίστροφο και το στέλνει στο επόμενο επίπεδο.

Τα πειράματα που θα πραγματοποιήσουμε μπορούν να χωριστούν ανάλογα με τον μηχανισμό που ελέγχουμε.

Για τον Rule Based μηχανισμό θα δώσουμε διαφορετικούς στόχους throughput και θα ελέγξουμε αν καταφέρνει να τον πετύχει καθώς και τις ενδιάμεσες καταστάσεις από τις οποίες περνάει. Επίσης για κάθε configuration θα μετρήσουμε τον χρόνο που κάνει για να προχωρήσει στο επόμενο καθώς και το μέσο throughput καθ' όλη τη διάρκεια.
Για τον Predictor θα μετρήσουμε το throughput του πραγματικού συστήματος για πιθανά configurations και θα ελέγξουμε αν οι αντίστοιχοι υπολογισμοί του προβλέπτη αντιστοιχούν στις πραγματικές τιμές. Έτσι θα μπορέσουμε να αναλύσουμε την συμπεριφορά του στο σύνολο των configurations.

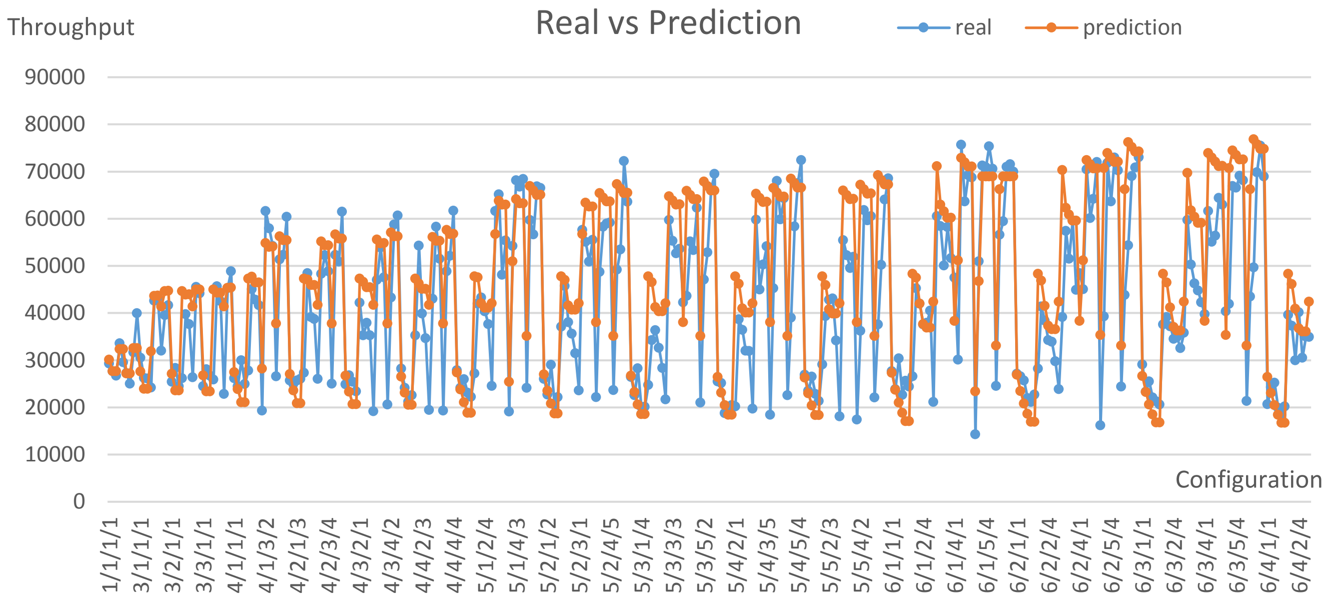
Τέλος θα ελέγξουμε τα high level constraints (time, money). Θα εκτιμηθεί η αποτελεσματικότητα των δύο μηχανισμών ως προς αυτά τα δύο constraints.

4.2 Ανάλυση των Αποτελεσμάτων

Η δομή που θα ακολουθήσουμε για την παρουσίαση των πειραμάτων είναι ανά τοπολογία. Για κάθε τοπολογία πρώτα σε σειρά θα είναι τα πειράματα του predictor, στη συνέχεια του rule based και τέλος η σύγκριση των δύο μηχανισμών.

4.2.1 WordCount Topology

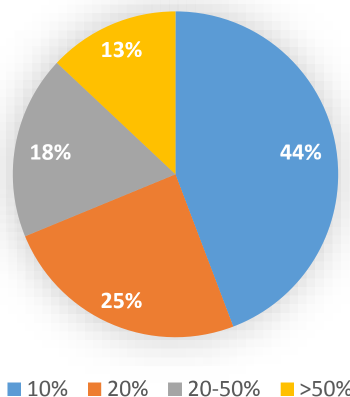
4.2.1.1 Predictor



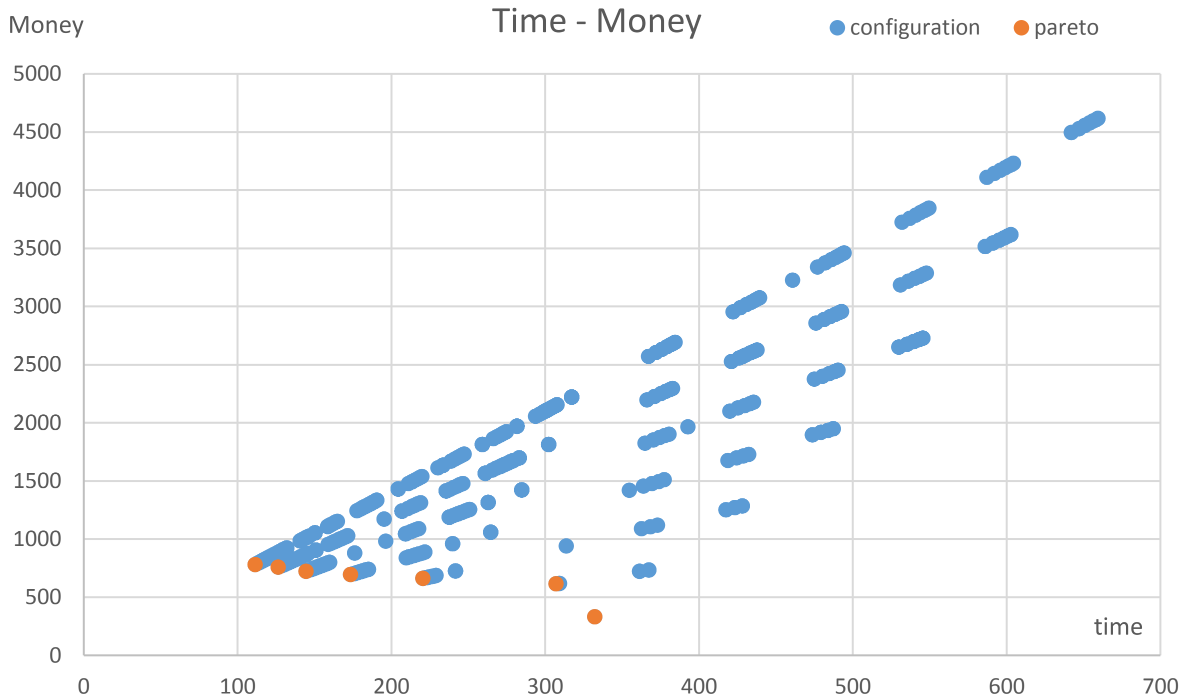
Η τοπολογία WordCount είναι απαιτητική τόσο από άποψη επεξεργασίας όσο και από άποψη δικτύου. Το σημαντικότερο εμπόδιο είναι το SplitSentence bolt που έχει την ευθύνη να χωρίσει την πρόταση που του έρχεται σε λέξεις αλλά και να μοιράσει(στείλει) τις λέξεις στα WordCount bolts. Γι' αυτό το λόγο αποτελεί και το μεγαλύτερο bottleneck της τοπολογίας. Για να μπορέσουμε να ανεβάσουμε την απόδοση της τοπολογίας θα πρέπει να παραλληλίσουμε το συγκεκριμένο component. Αν του δώσουμε επαρκή παραλληλία θα δούμε ότι η τοπολογία φτάνει σε ένα σημείο ισορροπίας, όπου μας δίνει το μέγιστο «τοπικό» throughput. Με τον όρο «τοπικό» throughput εννοούμε μια τιμή που μπορούμε να φτάσουμε με την πλειονότητα των configurations που λύνουν τα σημαντικά προβλήματα των bottlenecks. Για να καταφέρουμε να φτάσουμε στο μέγιστο δυνατό throughput (το οποίο μπορούμε να το πετύχουμε σε μικρότερο αριθμό configuration) θα πρέπει να εκμεταλλευτούμε την τοποθέτηση των components(spouts/bolts) ώστε να πετύχουμε καλύτερο locality και κατά συνέπεια μικρότερο network time. Αυτό όμως θέλει προσοχή, γιατί προσπαθώντας να κερδίσουμε από το locality μπορεί να υπερφορτώσουμε τους workers και να έχουμε προβλήματα ανταγωνισμού για τους πόρους.

Για την τοπολογία του WordCount μπορούμε να παρατηρήσουμε ότι πετυχαίνει πολύ καλές προβλέψεις για τα global minima και global maxima. Αυτό που μας ενδιαφέρει για τον μηχανισμό είναι να κάνει σωστές προβλέψεις κοντά στα global maxima έτσι ώστε να μπορούν να βρεθούν με ευκολία οι τοπολογίες που μας δίνουν το βέλτιστο throughput. Τα σημεία που οι προβλέψεις δεν είναι τόσο καλές είναι στα local minima. Σε αυτές τις περιπτώσεις ο μηχανισμός κάνει μια υπερεκτίμηση της επίδοσης και αυτό γιατί δεν μπορεί να υπολογιστεί

αποτελεσματικά ο χρόνος του επιπλέον network (συγκεκριμένα υπολογίζεται μικρότερο network time λόγω ενός μικρού locality και έτσι οδηγούμαστε στην υπερεκτίμηση). Τέλος, διαφορά στις προβλέψεις βλέπουμε και για συγκεκριμένα configurations που εμφανίζουν χαρακτηριστικά starvation, δηλαδή για τοπολογίες που έχουμε ένα component του worker χρησιμοποιεί την πλειονότητα των πόρων και τα υπόλοιπα components στον ίδιο worker γίνονται starve(π.χ. αν έχουμε στην τοπολογία μόνο ένα SplitSentence σε πολλούς workers, οι πόροι του worker που έχει αυτό το SplitSentence θα χρησιμοποιούνται κατά κόρον από αυτό). Σε τέτοιες περιπτώσεις το πραγματικό σύστημα δεν θα κάνει ολικό starve των υπόλοιπων component, ενώ το μοντέλο μας σε συγκεκριμένες οριακές περιπτώσεις μπορεί να οδηγηθεί σε τέτοια επιλογή.



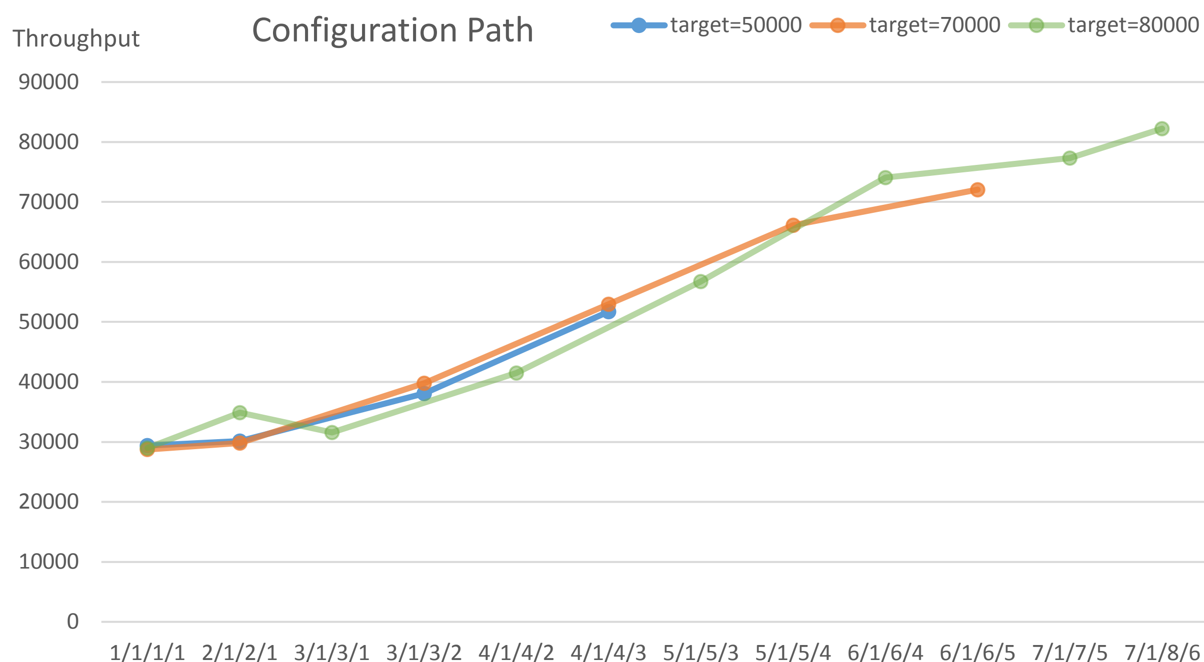
Μπορούμε να δούμε ότι η ακρίβεια του μηχανισμού για το WordCount topology είναι ικανοποιητική αλλά έχουμε έναν αριθμό προβλέψεων για το οποίο οι προβλέψεις έχουν μεγάλο ποσοστό αστοχίας. Συγκεκριμένα στο σύνολο 0-10% μαζί με ένα κομμάτι 10%+ έχουμε τα global minima/maxima καθώς και μερικές από τα ενδιάμεσα configurations. Στο σύνολο 10-20% έχουμε τα local maxima. Στο σύνολο 20-50%, κοντά στο 20%+ έχουμε τις περισσότερες ενδιάμεσες καταστάσεις. Τέλος, τη μεγαλύτερη αστοχία προβλέψεων την έχουμε για τα local minima και σε configuration που πέφτουν σε περίπτωση starvation.



Pareto Optimal Configurations (workers/spout/split/count)
1/1/1/1
2/2/2/2
3/3/3/3
4/4/4/2
5/5/5/2
6/6/6/3
7/7/7/3

Μπορούμε να παρατηρήσουμε ότι για μικρό αριθμό workers παίρνουμε το μέγιστο throughput για τοπολογίες που εκμεταλλεύονται το locality των operator, δηλαδή όταν έχουμε στον κάθε worker ένα component από κάθε είδος. Όσο αυξάνονται όμως οι workers η τοπολογία επηρεάζεται πολύ από το networking μεταξύ των SplitSentence bolts και των WorkCount bolts. Έτσι μπορούμε να δούμε ότι για περισσότερους workers παίρνουμε το μέγιστο throughput όταν έχουμε μεγάλο locality μεταξύ spout και SplitSentence και μικρό αριθμό WordCount(που σημαίνει και μικρό networking).

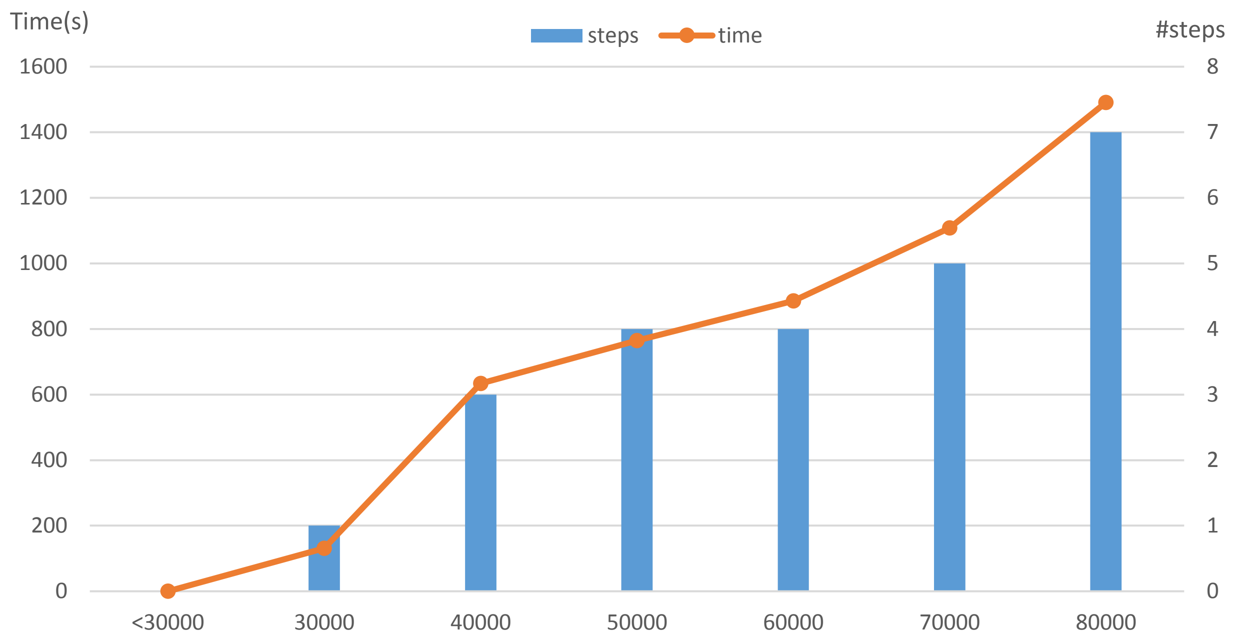
4.2.1.2 Rule Based



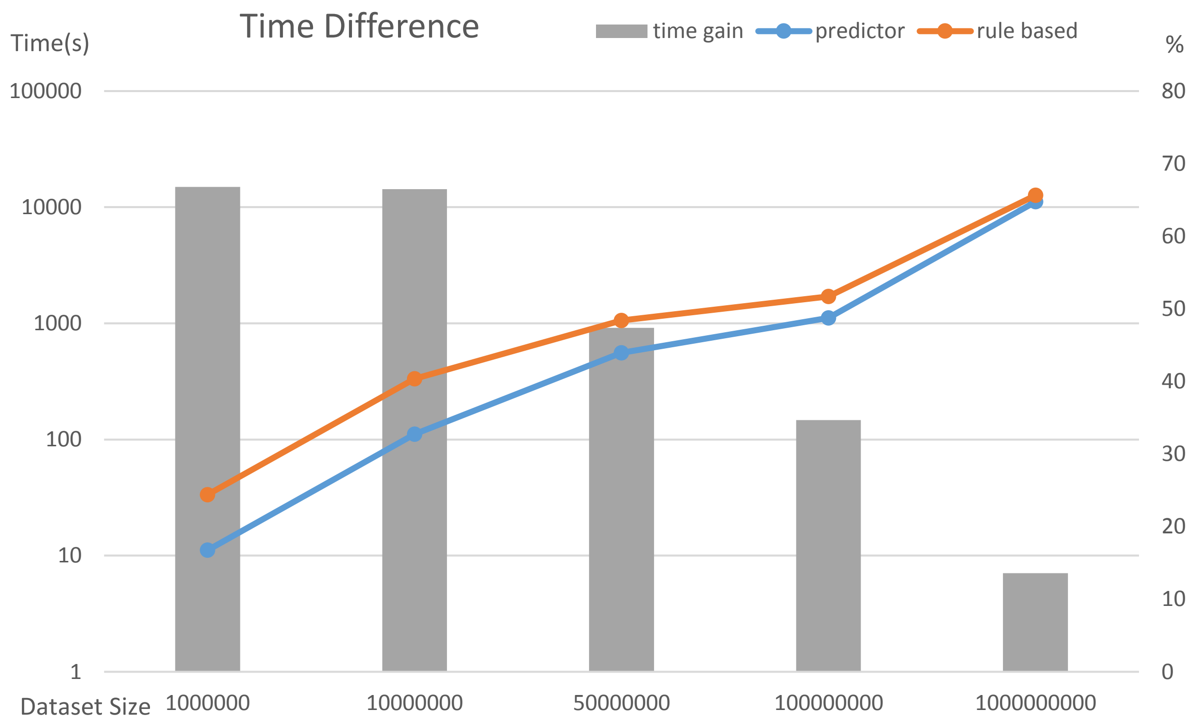
Στον μηχανισμό δώσαμε 3 διαφορετικούς και αυξανόμενους στόχους. Αυτό που μπορούμε να παρατηρήσουμε κατευθείαν είναι ότι σε αυτές τις 3 εκτελέσεις δεν πέρασε από ακριβώς τα ίδια ενδιάμεσα configuration, αλλά τυχαίνει να διαλέξει παραπλήσια configuration. Αυτό είναι λογικό και αναμενόμενο αφού ο μηχανισμός είναι real-time και αντιδρά στην κατάσταση που βρίσκεται κάθε στιγμή. Έτσι μπορεί να οδηγηθεί σε διαφορετική απόφαση για το configuration στο οποίο θα προχωρήσει. Συγκεκριμένα για throughput target 50000 και 70000 ακολούθησε την ίδια διαδρομή, ενώ για target 80000 ορισμένα από τα ενδιάμεσα configuration. Αυτό δεν οφείλεται σε καμία περίπτωση στον στόχο που το δόθηκε αλλά μόνο στην κατάσταση του συστήματος την στιγμή που ο μηχανισμός πήρε απόφαση για να αυξήσει την παραλληλία.

Ανάλογα με τον στόχο που θα δοθεί στον μηχανισμό αυξάνονται και τα βήματα που θα πρέπει να κάνει για να τον πετύχει. Όσο πιο μακριά βρίσκεται ο στόχος από την αρχική κατάσταση(δηλαδή όσο πιο μεγάλος είναι) τόσο περισσότερα βήματα θα εκτελεστούν και κατά συνέπεια μεγαλύτερος χρόνος.

Στόχος μας για τον συγκεκριμένο μηχανισμό είναι καθώς αυξάνεται ο στόχος τα βήματα και ο χρόνος που χρειαζόμαστε για αν το πετύχουμε αυξάνεται σχεδόν γραμμικά και όχι εκθετικά. Γι’ αυτό το λόγο οι αποφάσεις του rule based δεν γίνονται με κάποια προτεραιότητα αλλά μπορούν να παρθούν πολλές αποφάσεις μαζί. Αυτό μπορεί να το παρατηρήσει κανείς από τις μεταβάσεις των configurations όπου μπορούν να αυξηθούν σε ένα βήμα η παραλληλία πολλών components καθώς και οι workers.



4.2.1.3 Comparison



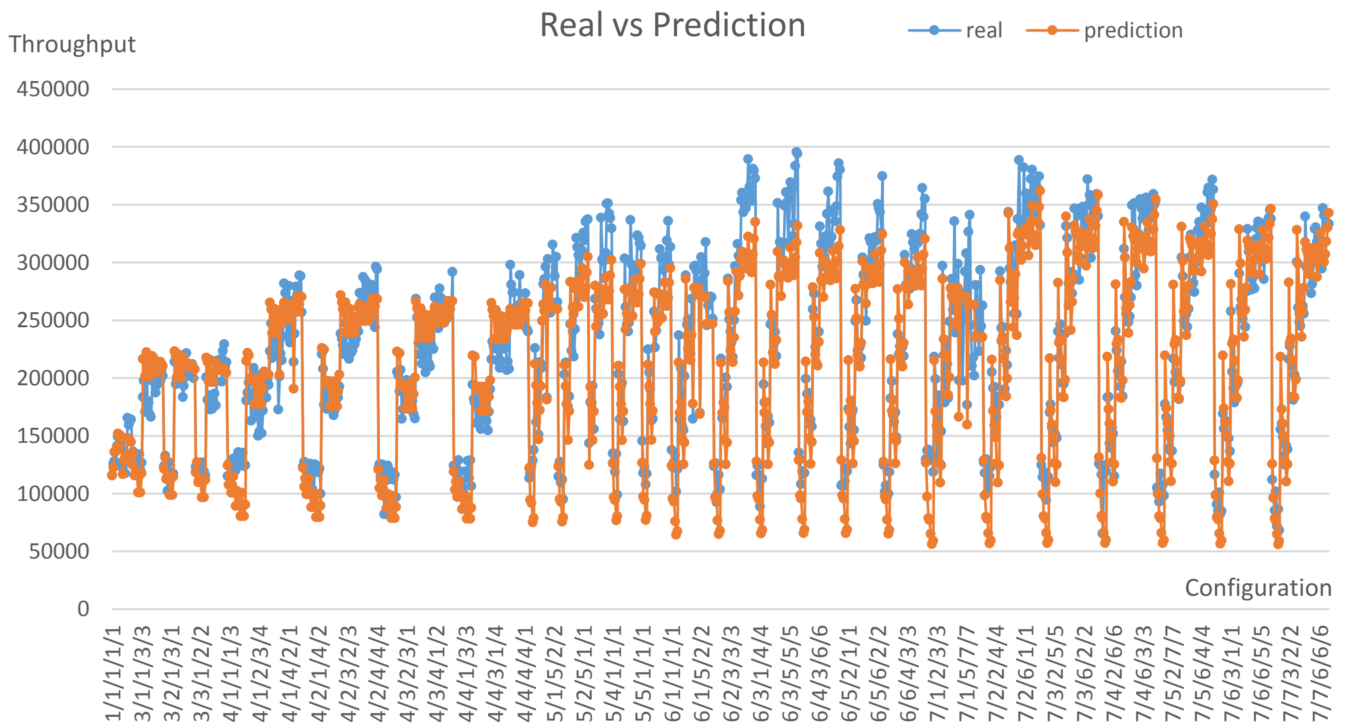
Συγκρίνοντας τους δύο μηχανισμούς από άποψη χρόνου μπορούμε να παρατηρήσουμε ότι από άποψη χρόνου κερδίζει σε όλες τις περιπτώσεις ο predictor. Αυτό είναι λογικό καθώς με τον predictor μπορούμε να βρούμε άμεσα το βέλτιστο configuration ενώ το rule based θα πρέπει να κάνει ένα είδους search και να περάσει από διαφορετικά configurations. Από το παραπάνω διάγραμμα

φαίνεται εύκολα ότι για τα μικρότερα datasets που έχουν και μικρότερο χρόνο εκτέλεσης ο predictor μας δίνει καλύτερα αποτελέσματα κατά 70%. Καθώς, όμως, αυξάνεται το dataset size αυξάνεται και ο χρόνος που θα πρέπει να παραμείνει running και σε αυτές τις περιπτώσεις το rule based αρχίζει να κερδίζει έδαφος καθώς προλαβαίνει να κάνει κάποια βήματα και να βελτιώσει την επίδοση του. Όπως είναι λογικό αν αυξηθεί πάρα πολύ το μέγεθος του dataset (καθώς τείνει στο άπειρο) το rule based θα προλάβει να φτάσει σε ένα βέλτιστο(ή σχεδόν βέλτιστο) configuration. Γι' αυτό και βλέπουμε μια σταδιακή μείωση του κέρδους καθώς αυξάνεται το dataset size.

Σε αυτό το σημείο μπορούμε να διακρίνουμε δύο περιπτώσεις σχετικά με τη διαφορά κέρδους. Στην πρώτη το rule based καταφέρνει να φτάσει σταδιακά το throughput που έχει δώσει και ο predictor και το dataset τείνοντας σε άπειρο μέγεθος το κέρδος σε χρόνο μεταξύ τους θα σταθεροποιηθεί σε απόλυτη τιμή ίση με τη διαφορά χρόνου που χρειάστηκε μέχρι να συγκλίνουν τα throughput των δύο μηχανισμών. Στη δεύτερη περίπτωση ο rule based μηχανισμός φτάνει σε ένα sub-optimal configuration και θα δίνει μικρότερο throughput σε σχέση με τον predictor για τη συνέχεια της εκτέλεσης. Έτσι καθώς το dataset size τείνει στο άπειρο η απόλυτη τιμή της διαφοράς χρόνου θα αυξάνεται συνεχώς και το ποσοστό κέρδους θα συγκλίνει σε μια σταθερή τιμή.

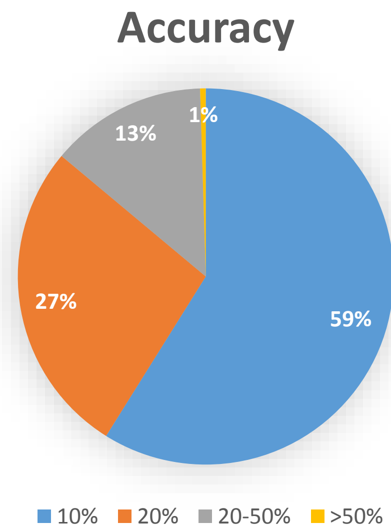
4.2.2 CyberShake Topology

4.2.2.1 Predictor

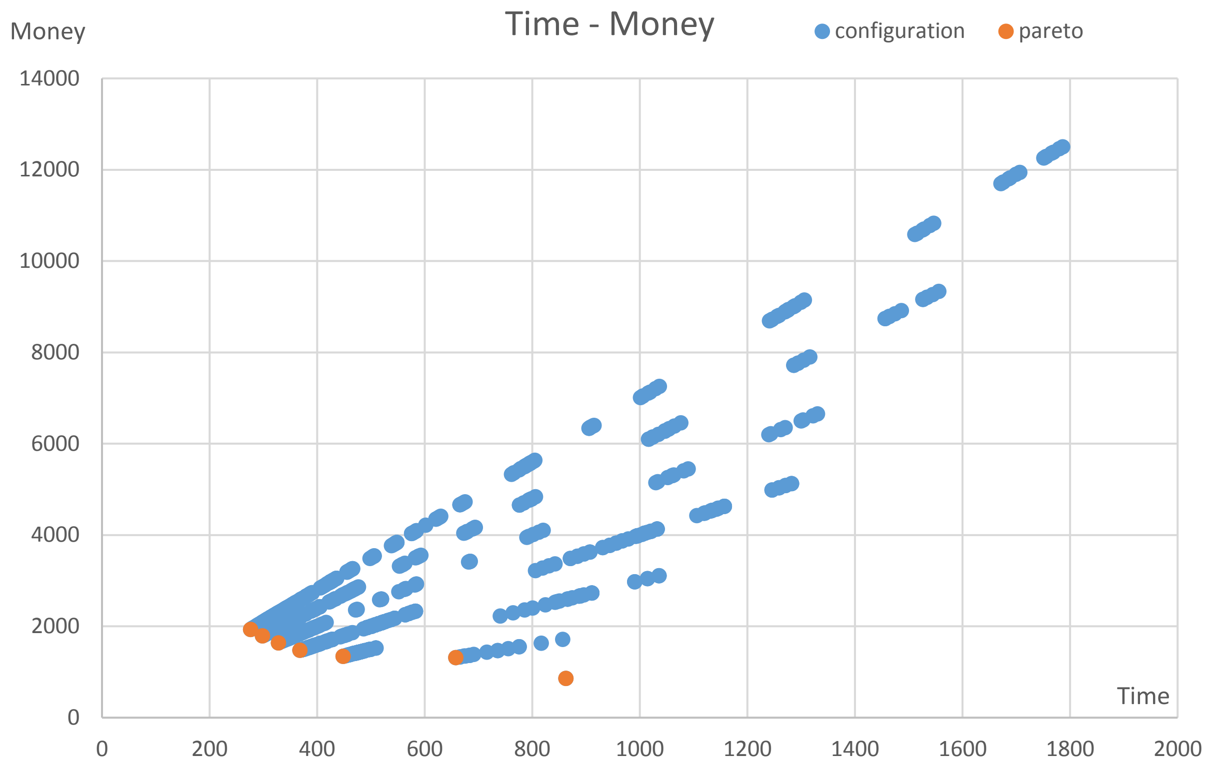


Η τοπολογία CyberShake αποτελείται από μη απαιτητικά components από άποψη επεξεργαστικής ισχύος. Έτσι το απαιτητικό σημείο της τοπολογίας είναι το Distributor από άποψη network, αφού στέλνει δύο streams προς τα general και general2. Επειδή αναφέρουμε το πρόβλημα του network δεν σημαίνει ότι δεν έχει αντίκτυπο και στο cpu time, το αντίθετο μάλιστα όσο περισσότερο networking δημιουργείτε τόσο περισσότερους πόρους καταναλώνουμε. Η αύξηση στο throughput της τοπολογίας έρχεται όπως είναι αναμενόμενο δίνοντας παραλληλία στο Distributor. Για να πετύχουμε, όμως, το μέγιστο θα πρέπει να βρούμε τις τοπολογίες που προσφέρουν την βέλτιστη ισορροπία(δηλαδή ικανοποιητικό locality και ταυτόχρονα μειωμένο networking) μεταξύ spout-Distributor και Distributor-General/General2.

Για την τοπολογία CyberShake παρατηρούμε ότι πετυχαίνει να ακολουθεί το outline των πραγματικών μετρήσεων με μεγάλη ακρίβεια. Συγκεκριμένα, δίνει ακριβείς προβλέψεις για τις μεταβατικές καταστάσεις και κοντά στα global maxima. Για συγκεκριμένες περιοχές από global maxima ακολουθεί το μέγιστο throughput αλλά δίνει περισσότερη υποεκτίμηση με αποτέλεσμα να μένει πιο μακριά από την πραγματική τιμή. Στα global minima και σε μερικά από τα local minima υποεκτιμά την επίδοση παραπάνω από την πραγματική τιμή αλλά οι συγκεκριμένες περιοχές δεν είναι περιοχές ενδιαφέροντος(μάλιστα η επιπλέον υποεκτίμηση βοηθάει ένα search space να απομακρυνθεί από αυτές τις περιοχές και να κινηθεί προς τα maxima)



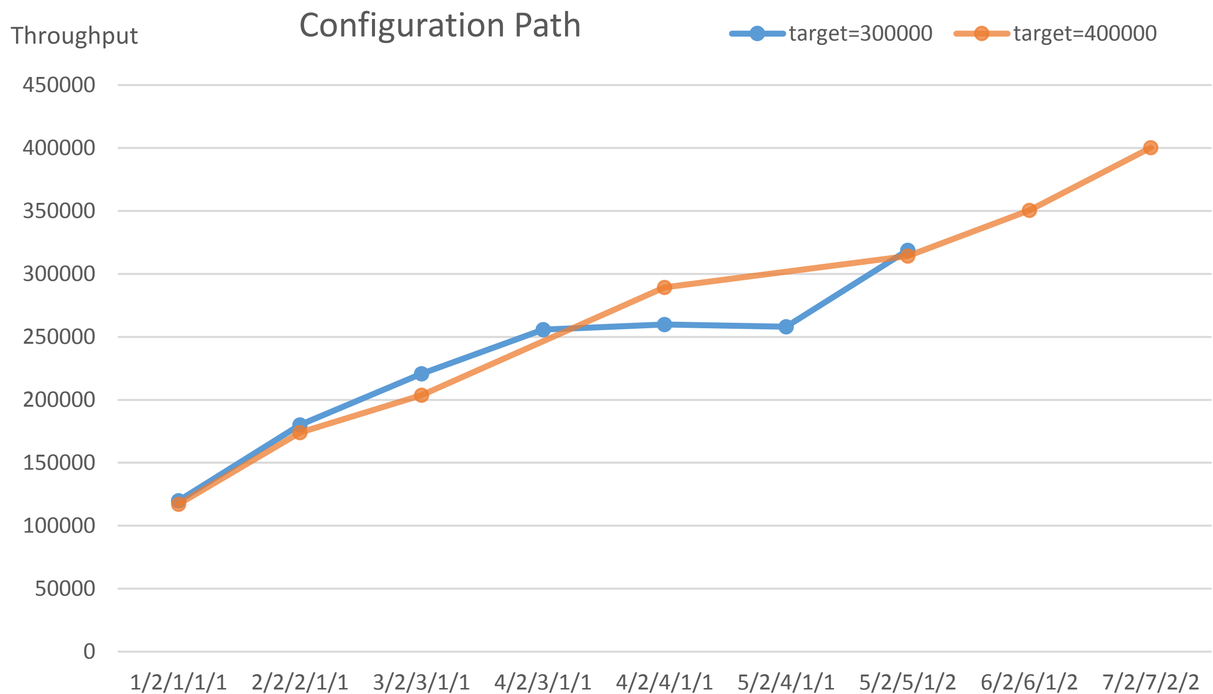
Η ακρίβεια του μηχανισμού στο CyberShake topology είναι πολύ καλή καθώς στην πλειονότητα των configurations είναι πολύ κοντά στην πραγματική τιμή. Το σύνολο 0-10% αποτελείται από τις ενδιάμεσες καταστάσεις καθώς και τις περιοχές των global maximum. Το σύνολο 10-20% αποτελείται τις ορισμένες οριακές περιπτώσεις των global maximum καθώς και μερικές περιπτώσεις local maximum. Τέλος το σύνολο 20-50% αποτελείται από τις περιοχές των global minimum.



Pareto Optimal Configurations (workers/spout/distributor/general/general2)
1/1/1/1/1
2/1/2/1/1
3/2/2/1/1
4/2/3/1/1
5/2/5/5/5
6/2/6/6/6
7/2/7/7/7

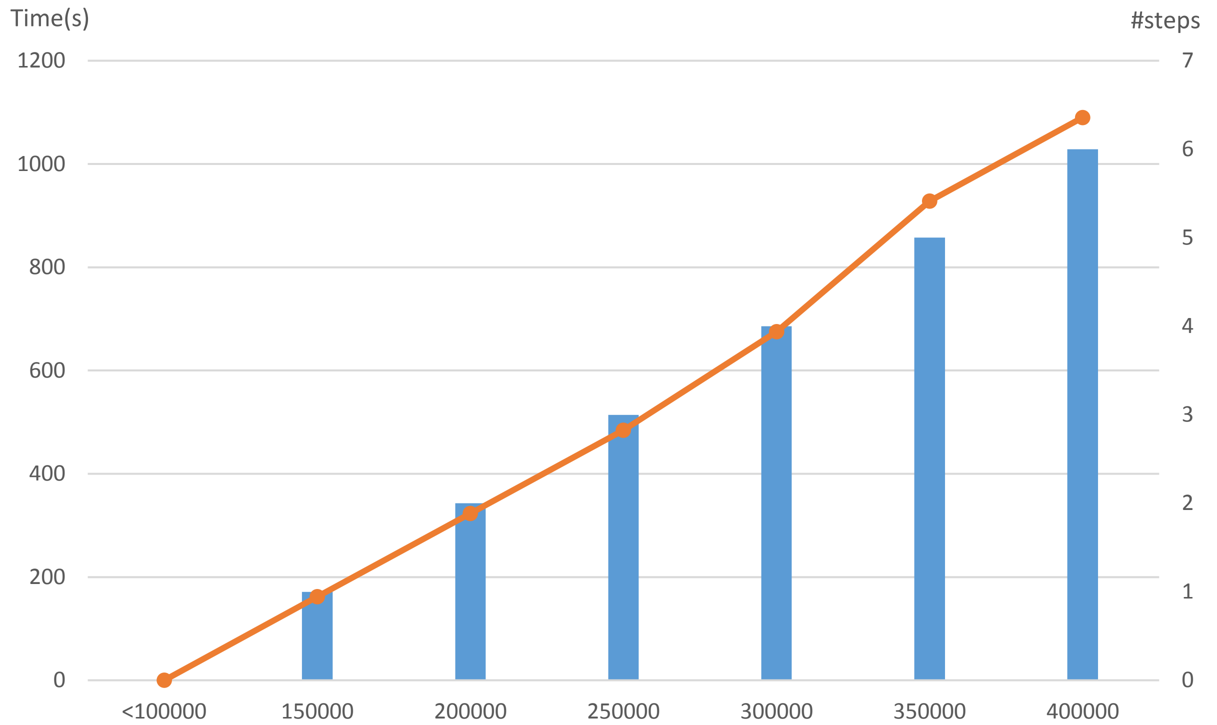
Από τα αποτελέσματα βλέπουμε ότι για μικρό αριθμό από workers πετυχαίνουμε το μέγιστο throughput δίνοντας παραλληλία στο Distributor και κρατώντας χαμηλή την παραλληλία των υπόλοιπων components(επειδή έχουμε λίγους workers και μόνο με την παραλληλία του Distributor πετυχαίνουμε ικανοποιητικό locality). Όσο αυξάνουμε τους workers χρειάζεται να προσθέσουμε και ισόποσο αριθμό από τα General και General2. Αυτό δημιουργεί αρκετό networking αλλά παράλληλα και πολύ υψηλό locality και δεδομένου ότι τα δύο τελικά components είναι ιδιαίτερα ελαφριά από άποψη χρήσης πόρων δεν δημιουργούν πτώση της επίδοσης λόγω ανταγωνισμού πόρων. Τέλος ο αριθμός των spouts δεν παίζει ιδιαίτερο ρόλο καθώς μπορούν να παράγουν και να στείλουν πολύ γρήγορα.

4.2.2.2 Rule Based

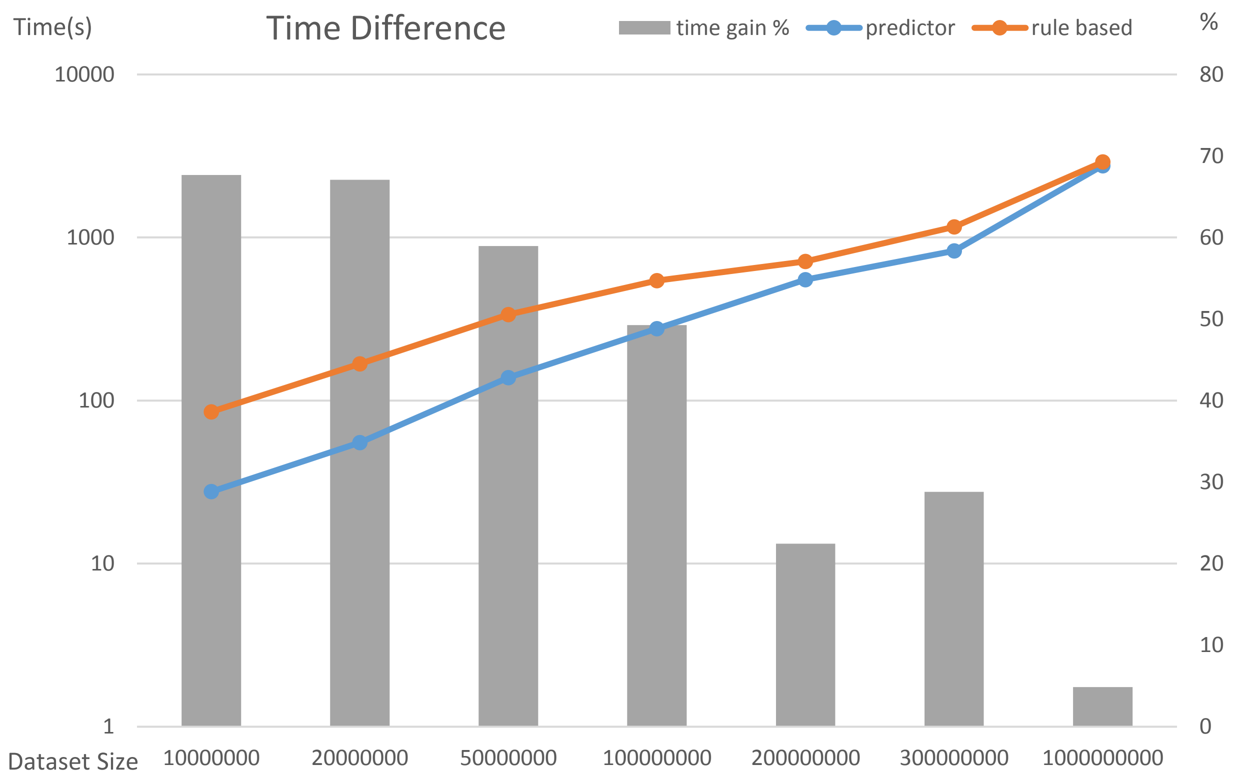


Στην τοπολογία CyberShake το κύριο bottleneck όπως εξηγήσαμε και στο κομμάτι του predictor είναι το Distributor. Βλέπουμε ότι το rule based ασχολείται κατά κύριο λόγο με την παραλληλοποίηση του συγκεκριμένου component αφού τα ενδιάμεσα configurations επηρεάζουν μόνο αυτό. Για το λόγο αυτό μπορούμε να δούμε ότι τα paths που ακολουθούνται είναι παρόμοια με μία μόνο διαφορά. Για throughput target 300000 τα μόνα βήματα που γίνονται είναι για να αυξηθεί η παραλληλία του distributor και των worker, μόνο στο τελευταίο βήμα λαμβάνεται υπόψιν και το General2 και του δίνεται παραπάνω παραλληλία. Για στόχο 400000 που είναι σαφώς μεγαλύτερος και φτάνει τα όρια της τοπολογίας βλέπουμε ότι προς τα τελευταία βήματα λαμβάνονται υπόψιν και τα General/General2 για να δημιουργηθεί ενός είδους locality και να μπορέσει η τοπολογία να φτάσει σε τόσο υψηλό throughput.

Για τα βήματα, στο CyberShake οι αποφάσεις παίρνονται αρκετά γρήγορα για το πως θα προχωρήσει ο μηχανισμός και έτσι βλέπουμε να απαιτείται γραμμικός χρόνος μεταξύ των steps.



4.2.2.3 Comparison

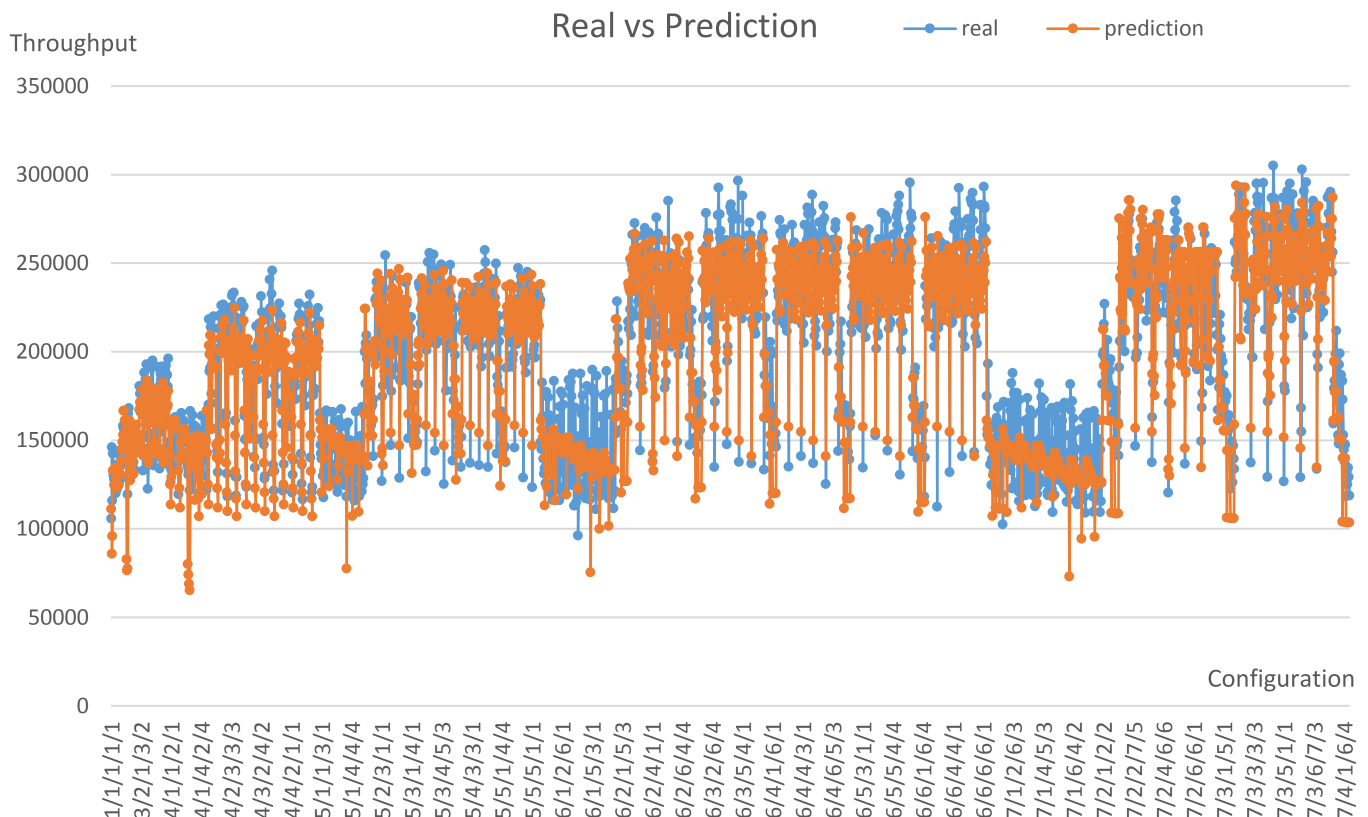


Ο predictor και πάλι μας δίνει σαφές κέρδος στην πλειονότητα των μεγεθών των datasets και μόνο όταν αυξάνεται σημαντικά μπορούμε να δούμε ότι η διαφορά

που μας δίνει μπορεί να μην ενδιαφέρει. Ένα μικρό άλμα στο κέρδος από 20% σε 28% εμφανίζεται γιατί το rule based χρειάστηκε περισσότερο χρόνο για να προχωρήσει σε απόφαση να δώσει παραλληλία και έτσι το step «κόστισε» παραπάνω.

4.2.3 Montage Topology

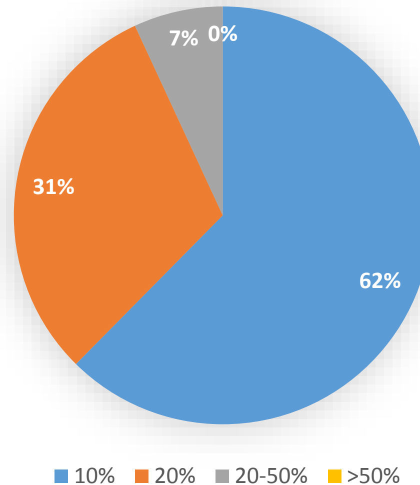
4.2.3.1 Predictor



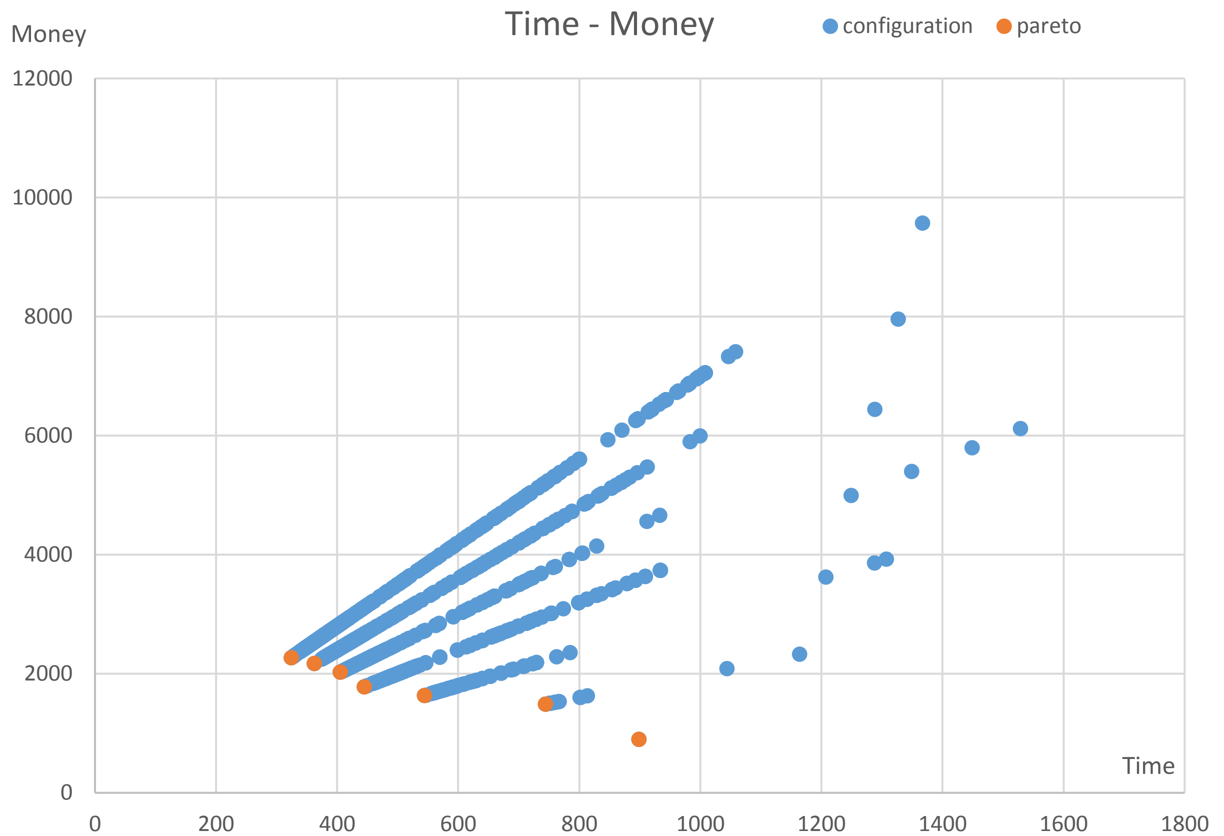
Η τοπολογία Montage είναι πιο απαιτητική τόσο σχεδιαστικά όσο και από άποψη component. Συγκεκριμένα, το component General είναι απαιτητικό από άποψη network και το Aggregator από άποψη επεξεργασίας. Για να δούμε αύξηση στο throughput της τοπολογίας θα πρέπει να παραλληλίσουμε κατάλληλα τα δύο προηγούμενα components. Για να πετύχουμε το μέγιστο throughput θα πρέπει να βρούμε τη βέλτιστη ισορροπία ώστε να εξασφαλίσουμε τέτοια τοποθέτηση για το Aggregator ώστε να έχει καλό locality με όλα τα υπόλοιπα components (θέλει προσοχή γιατί το σημαντικότερο είναι να έχουμε locality μεταξύ spout-General-Aggregator που είναι τα βασικότερα ενώ το General2 έρχεται με δεύτερη προτεραιότητα).

Για την τοπολογία Montage βλέπουμε ότι ακολουθεί με ακρίβεια το outline των πραγματικών τιμών και οι προβλέψεις είναι γενικότερα ακριβείς. Την μεγαλύτερη

ακρίβεια την πετυχαίνουμε για τις ενδιάμεσες καταστάσεις και σε περιοχές από global/local maxima, ενώ μικρότερη ακρίβεια έχουμε για περιοχές από global/local minima τις οποίες υποτιμούμε.



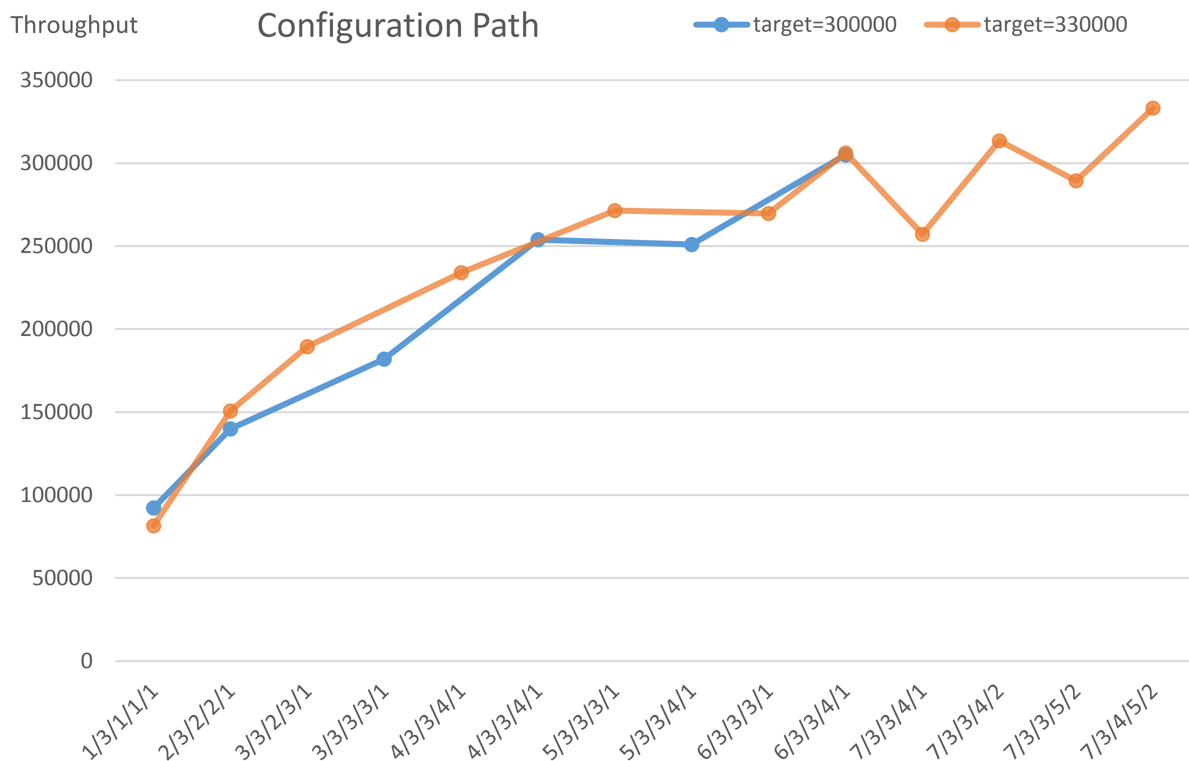
Η ακρίβεια του συστήματος για το montage topology είναι συνολικά εξαιρετική με πολύ ακριβείς προβλέψεις. Στο σύνολο 10-20% εμπεριέχεται η πλειονότητα των προβλέψεων, μεταξύ των οποίων είναι οι περιοχές των ενδιάμεσων configuration καθώς και τα global/local maxima. Στο σύνολο 10-20% εμπεριέχονται κοντά στο 10%+ μερικές περιοχές από global maxima μαζί με τις μεταβατικές περιοχές κοντά σε local minima/maxima. Τέλος το σύνολο 20-50% περιλαμβάνει τις περιοχές από global minima καθώς και περιοχές από local minima.



Pareto Optimal Configurations (workers/spout/general/aggregator/general2)
1/1/1/1/1
2/1/2/1/2
3/2/2/3/3
4/2/3/4/4
5/2/4/5/5
6/5/2/2/1
7/4/2/6/4

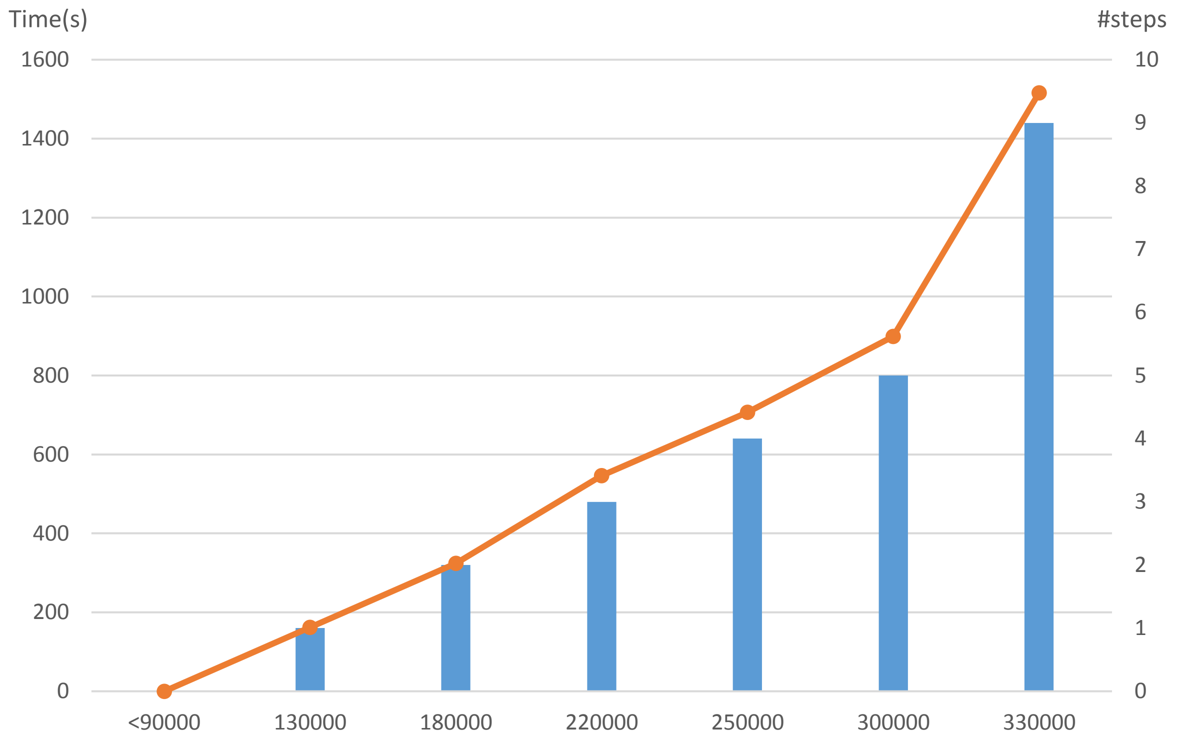
Μπορούμε να δούμε εύκολα ότι σε όλες τις περιπτώσεις παίζει ιδιαίτερη σημασία το locality του Aggregator με τα υπόλοιπα components, καθώς είναι το κέντρο της τοπολογίας και όλα περνάνε από αυτό το component. Για μικρότερο αριθμό workers δίνεται έμφαση τόσο στην παραλληλία των General, Aggregator όσο και του General2, για μεγαλύτερο αριθμό, όμως, μπορούμε να δούμε ότι θυσιάζεται η παραλληλία του General2 για να έχουμε μικρότερο network παρόλο που πρόκειται για ένα αρκετά ελαφρύ component.

4.2.3.2 Rule Based

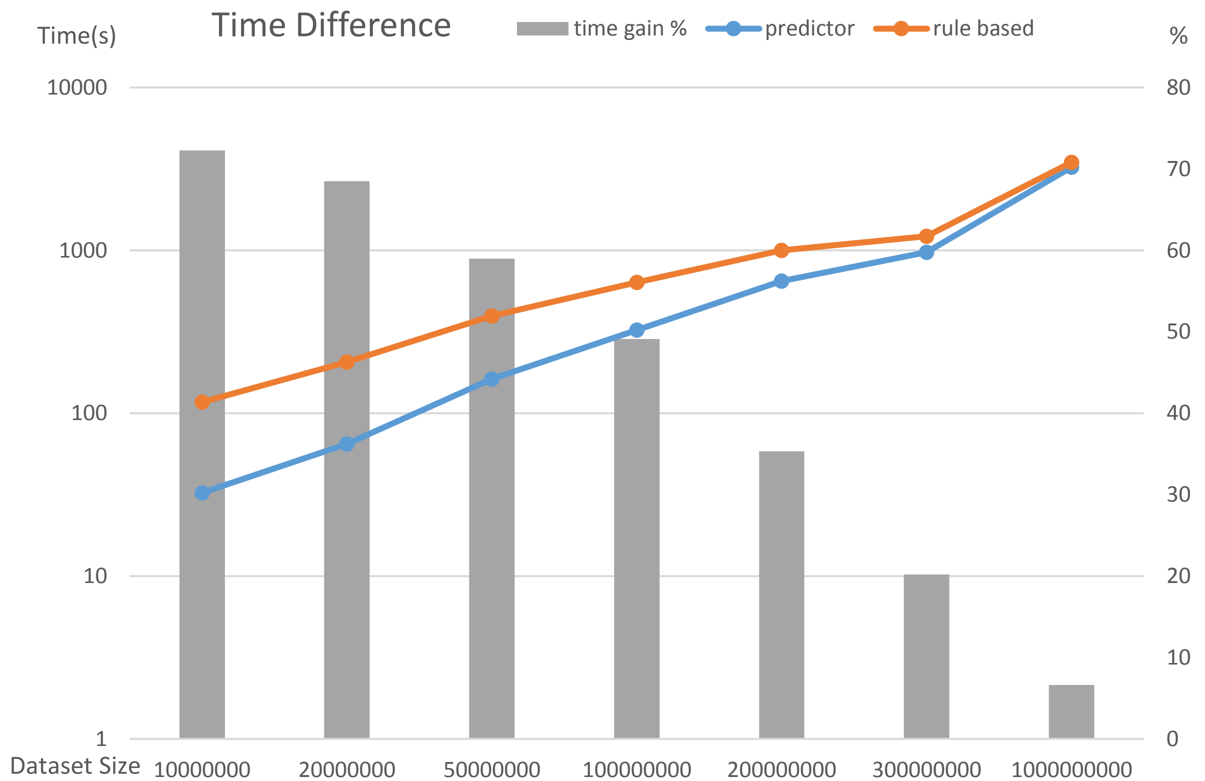


Η τοπολογία Montage αποτελεί πιο ιδιαίτερη και δύσκολη περίπτωση αφού εμφανίζει δύο bottlenecks, τα General και Aggregator. Όταν του δώσαμε ένα στόχο που μπορεί να πετύχει σχετικά εύκολα (μπορεί ο στόχος 300000 να είναι υψηλός αλλά μπορεί αν επιτευχθεί από μεγάλο αριθμό configurations) εκτέλεσε βήματα προς συγκεκριμένη κατεύθυνση και πέτυχε γρήγορα τον στόχο του (η κατεύθυνση είναι να δώσει workers και μαζί να αυξήσει την παραλληλία όσο χρειάζεται στα bottlenecks που συναντάει). Όταν όμως του δώσαμε στόχο 330000 που είναι κοντά στα όρια της τοπολογίας χρειάστηκε να περάσει από αρκετές μη βέλτιστα configuration και να εκτελέσει και παραπάνω βήματα. Συγκεκριμένα για να δώσουμε αύξηση 30000(10%) χρειάστηκαν 4 παραπάνω βήματα.

Όσο για τον χρόνο που χρειάζεται για να πετύχει τον επιθυμητό στόχο είναι και εδώ γραμμικός με εξαίρεση την οριακή περίπτωση που χρειάστηκε επιπλέον βήματα και κατά συνέπεια και παραπάνω χρόνο.



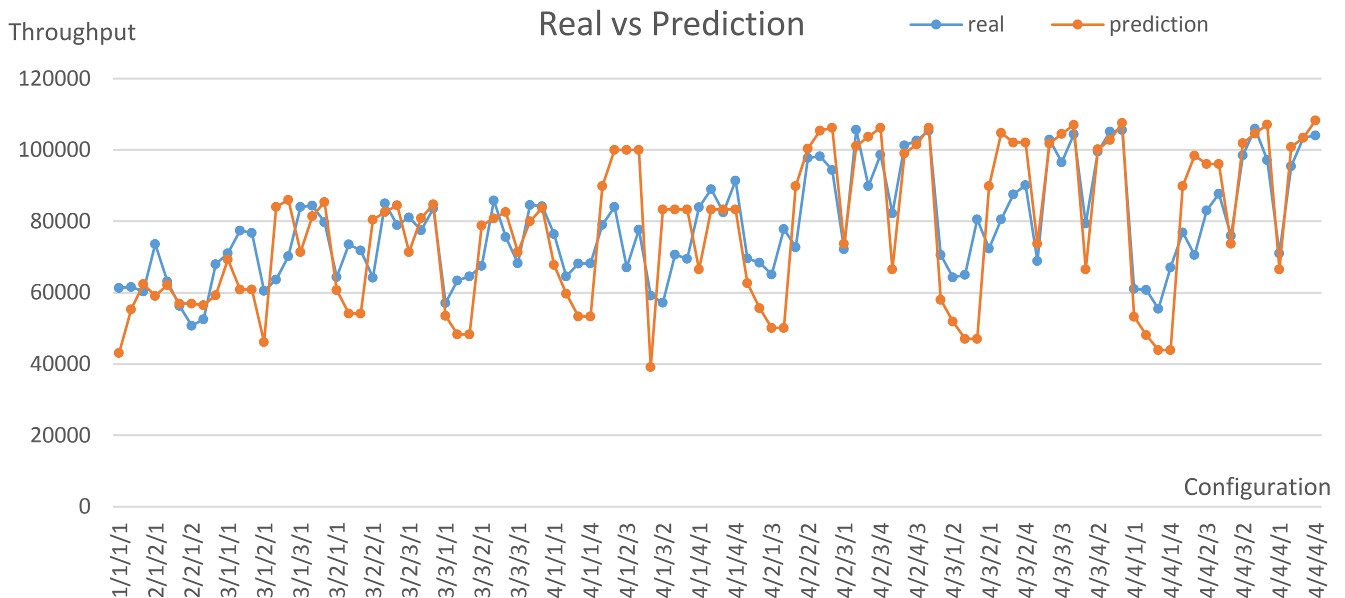
4.2.3.3 Comparison



Το κέρδος που παίρνουμε από τον predictor είναι συνολικά μεγαλύτερο για το montage και αυτό γιατί είναι γενικά πιο ακριβής στις προβλέψεις του σε σχέση με τις υπόλοιπες τοπολογίες. Φυσικά όπως και στις προηγούμενες το κέρδος πέφτει όσο αυξάνεται το dataset size.

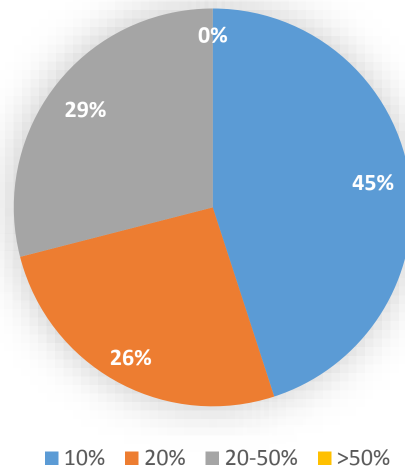
4.2.4 Matrix Topology

4.2.4.1 Predictor

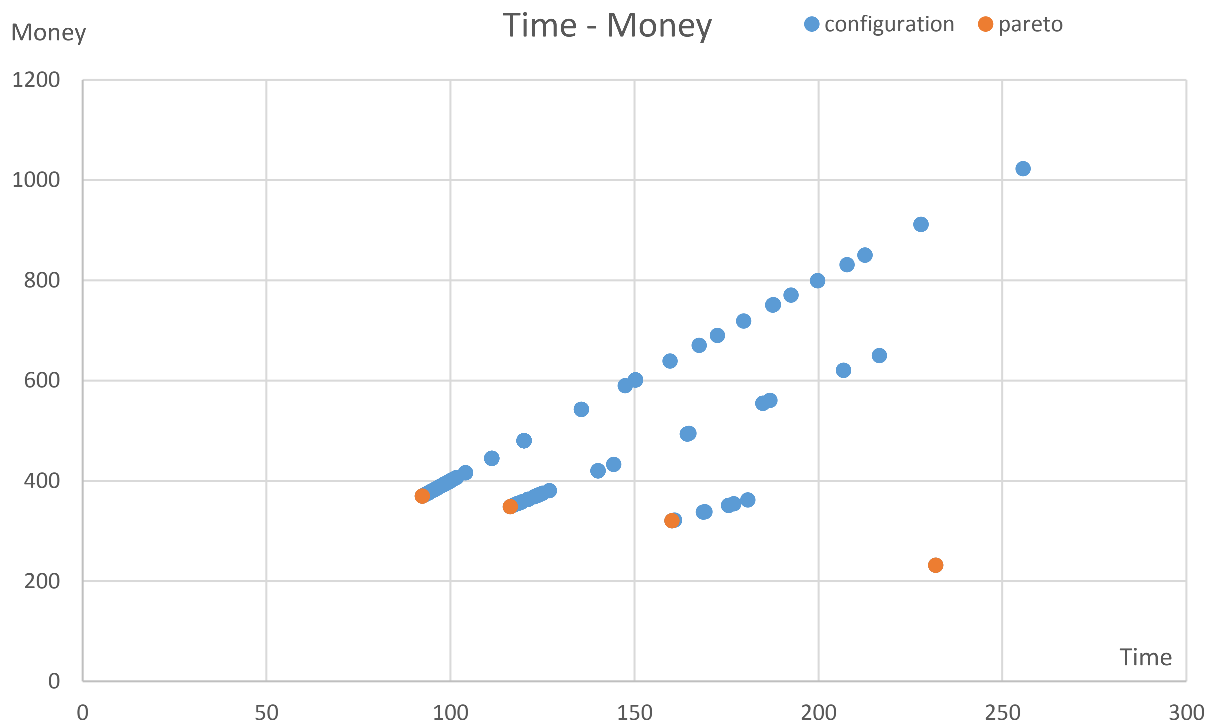


Η τοπολογία Matrix είναι cpu intensive και σημαντικά bottlenecks είναι τόσο ο MatrixCreator όσο και ο MatrixInverter. Για να πετύχουμε το μέγιστο throughput θα πρέπει να δώσουμε κατάλληλη παραλληλία σε αυτά τα δύο components.

Οι προβλέψεις για την τοπολογία Matrix ακολουθούν το outline των πραγματικών τιμών, όμως αλλάζουν από υποεκτίμηση σε υπερεκτίμηση ανάλογα την περιοχή. Συγκεκριμένα υποεκτιμά τις περιοχές με global/local minima(προβλέπεται μεγαλύτερο network penalty από το πραγματικό) και δίνει μια παραπάνω υπερεκτίμηση σε σημεία από local minima(συμβαίνει λόγω starvation όπως και στο WordCount). Από την άλλη ακολουθεί με ακρίβεια τις μεταβατικές καταστάσεις και τα global maxima(που μας ενδιαφέρουν περισσότερο)



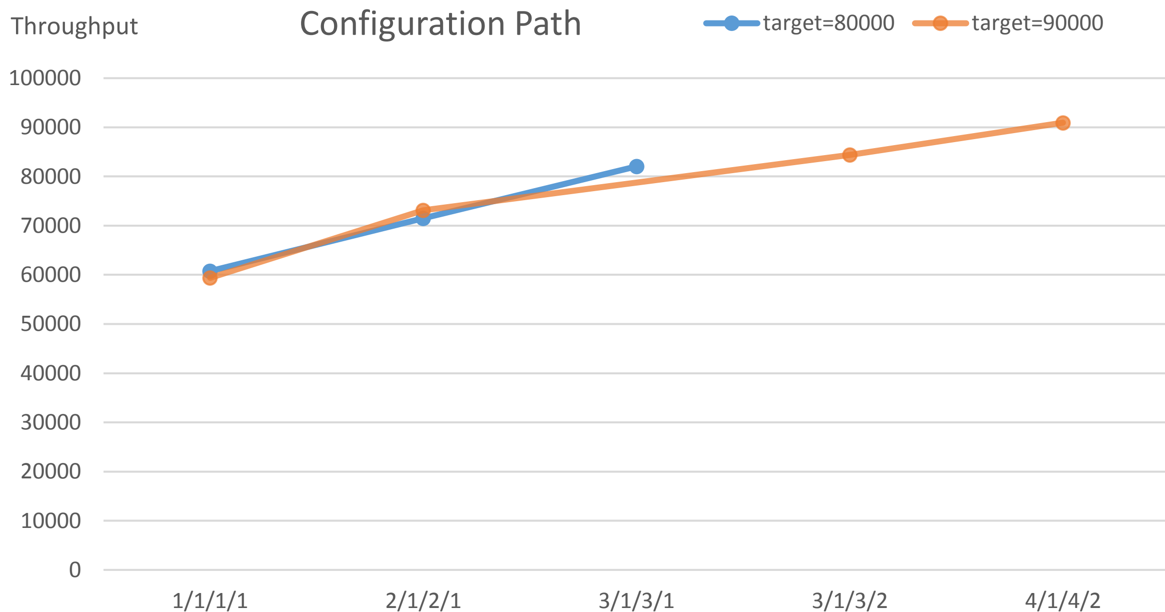
Η ακρίβεια του συστήματος για την τοπολογία Matrix είναι ικανοποιητική. Το σύνολο 0-10% αποτελείται από τις περιοχές global maximum καθώς και τις μεταβατικές περιοχές. Το σύνολο 10-20% αποτελείται από τις περιοχές local maximum. Τέλος το σύνολο 20-50% αποτελείται από τις περιοχές global/local minimum.



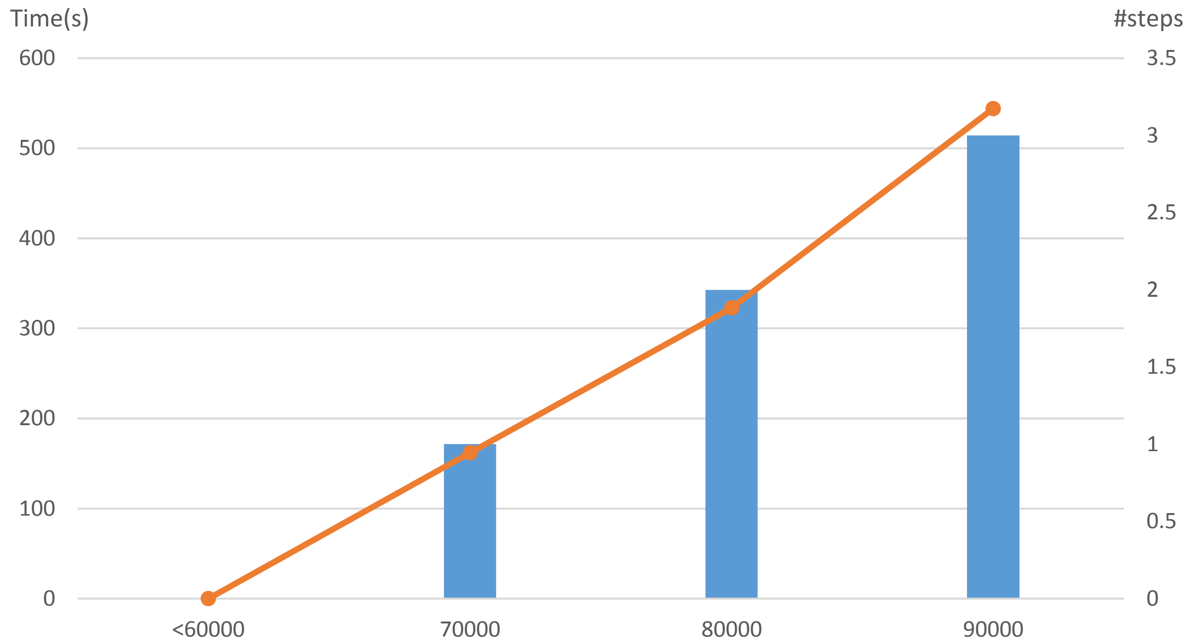
Pareto Optimal Configurations (workers/spout/creator/inverter)
1/1/1/1
2/1/1/2
3/1/2/3
4/4/4/4

Παρατηρούμε ότι για μικρό αριθμό worker μαζί με τον παραλληλισμό των MatrixCreator και MatrixInverter λύνεται και το πρόβλημα του locality. Καθώς, όμως, ανεβαίνει ο αριθμός των worker το locality αρχίζει να έχει μεγαλύτερη σημασία.

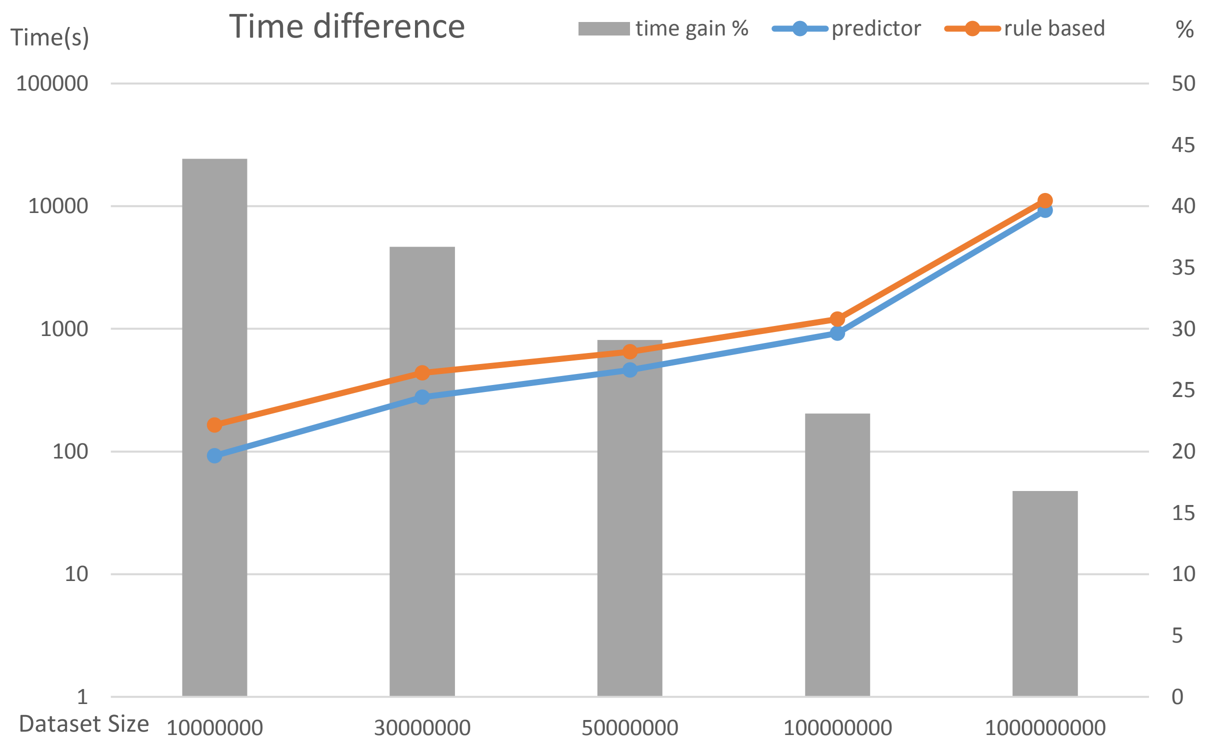
4.2.4.2 Rule Based



Όπως και στις προηγούμενες τοπολογίες ο μηχανισμός κάνει γραμμικό χρόνο για την μετάβαση από το ένα configuration στο επόμενο. Τα ενδιάμεσα configuration μπορεί να είναι διαφορετικά αλλά σε κάθε περίπτωση πετυχαίνει τον επιθυμητό στόχο.



4.2.4.3 Comparison



Μπορούμε γρήγορα να παρατηρήσουμε ότι σε σχέση με τις προηγούμενες περιπτώσεις εδώ το ποσοστό που κερδίζει ο predictor δεν μειώνεται με τον ίδιο γρήγορο ρυθμό. Αυτό συμβαίνει γιατί το rule based φτάνει σε ένα μη βέλτιστο configuration που χάνει σε σχέση με την αντίστοιχη πρόβλεψη. Έτσι οι δύο μηχανισμοί δεν θα φτάσουν σε κατάσταση που θα δίνουν το ίδιο throughput αλλά

θα υπάρχει μια διαφορά μεταξύ τους, με τον predictor να κερδίζει συνεχώς(μικρά κομμάτια κέρδους που αθροίζονται και συντηρούν τη διαφορά κέρδους).

4.3 Συμπεράσματα

Για τον predictor μπορούμε να συμπεράνουμε τα εξής για την ακρίβειά του :

- Υπάρχουν πολλές παράμετροι που μπορούν να επηρεάσουν τις επιδόσεις του πραγματικού συστήματος και είναι δύσκολο να μοντελοποιηθούν. Για παράδειγμα ο ανταγωνισμός για τους πόρους του worker μεταξύ των threads και το context switching που μπορεί να συμβαίνει επηρεάζουν αρνητικά τις επιδόσεις.
- Η σωστή πρόβλεψη του network παίζει πολύ σημαντικό ρόλο στην επίδοση του συστήματος. Components με πολλά εξερχόμενα streams δημιουργούν bottlenecks στις περισσότερες περιπτώσεις. Γι' αυτό το λόγο πολύ σημαντικό ρόλο παίζει το locality γειτονικών component(δηλαδή να εκτελούνται στον ίδιο worker).
- Είναι πολύ σημαντικό να βρεθεί το configuration που προσφέρει την καλύτερη δυνατή ισορροπία μεταξύ παραλληλίας, network και locality. Αυτό είναι που θα μας δώσει την βέλτιστη επίδοση.
- Βέλτιστη επίδοση δεν είναι απαραίτητο ότι παίρνουμε μόνο από ένα μοναδικό configuration. Υπάρχει ένα σύνολο από configurations που μας δίνουν βέλτιστη επίδοση.
- Σφάλματα που μπορούν να δημιουργηθούν είτε από τις αρχικές μετρήσεις(επειδή γίνονται με δειγματοληψία 5%) προσθέτουν αβεβαιότητα στα μοντέλα που δημιουργούμε. Σαν επέκταση, οι προβλέψεις που παράγει το WEKA επηρεάζονται σημαντικά από το μοντέλο και τον αλγόριθμό που χρησιμοποιείται.

Η ακρίβεια σε έναν μηχανισμό πρόβλεψης είναι πρωτεύουσας σημασίας, όμως είναι χρήσιμο να αναρωτηθούμε που χρειαζόμαστε αυτή την ακρίβεια. Οι περιοχές που χρειάζεται να είναι ακριβής ο μηχανισμός είναι τα global maxima για να μπορεί να διακρίνει λεπτές διαφορές μεταξύ των configuration της περιοχής(και μάλιστα να ακολουθεί τις πραγματικές με μια μικρή υποτίμηση της επίδοσης για να μην δίνει false positives). Αντίθετα οι περιοχές των global και local minimum δεν μας ενδιαφέρουν ιδιαίτερα αφού δεν μας χρησιμεύουν, έτσι μπορούμε να δώσουμε μια μεγαλύτερη ελαστικότητα σε αυτές τις περιοχές.

Για τον rule based μηχανισμό έχουμε να παρατηρήσουμε τα εξής :

- Καταφέρνει αποτελεσματικά να πραγματοποιεί πολλές κινήσεις παραλληλίας σε ένα βήμα (δίνει παραλληλία σε πολλαπλά components και σε workers ταυτόχρονα). Αυτό του δίνει το πλεονέκτημα ότι μπορεί να πετύχει συνεχώς αυξανόμενους στόχους με σχεδόν γραμμική αύξηση του χρόνου που χρειάζεται.

- Ανάλογα με την «επιμονή» που του δίνουμε (κατεβάζουμε το όριο για το severity) μπορεί να κάνει τον μηχανισμό πιο γρήγορα αλλά ταυτόχρονα και πιο απρόσεκτο στα βήματα που εκτελεί.
- Για να πετύχει ένα στόχο που του έχουμε δώσει σε δύο διαφορετικές εκτελέσεις μπορεί αν περάσει από διαφορετικά ενδιάμεσα configurations και αυτό γιατί την στιγμή που πήρε την απόφαση οι μετρήσεις των διάφορων components είναι διαφορετικές μεταξύ των εκτελέσεων και στη μια να επιλέξει να δώσει παραλληλία σε ένα και στην επόμενη σε περισσότερα.
- Αν του δώσουμε στόχο που δεν μπορεί να φτάσει θα συνεχίσει να ψάχνει με μικρό αλλά υπαρκτό κίνδυνο να χαλάσει επίδοση που έχει πετύχει.

Συγκρίνοντας τους δύο μηχανισμούς μπορούμε να θέσουμε τα εξής :

- Ο predictor είναι μπορεί να μας δώσει με ακρίβεια το configuration που μας δίνει την βέλτιστη επίδοση άμεσα και γρήγορα, αλλά χρειάζεται προεργασία για την δημιουργία των μοντέλων. Από την άλλη ο rule based τρέχει παράλληλα με την τοπολογία και την αλλάζει όταν κρίνει ότι χρειαστεί χωρίς να χρειάζεται προηγούμενη γνώση για την συμπεριφορά της.
- Το rule based δεν έχει καμία γνώση για την τοπολογία που πρέπει να παραλληλίσει ούτε κρατάει γνώση που έχει αποκτήσει, έτσι κάθε φορά τρέχει με την ίδια λογική. Από την άλλη με τον predictor Μπορούμε να κρατήσουμε μέσω των μοντέλων γνώση που έχουμε αποκτήσει για τα components τοπολογιών. Έτσι μπορούμε να δημιουργήσουμε βιβλιοθήκες με τέτοια μοντέλα που θα μπορούν τόσο να επεκταθούν όσο και να επαναχρησιμοποιηθούν από άλλες τοπολογίες.
- Με τον predictor μπορεί να γίνει ένας προσχεδιασμός πριν βάλει κάποιος την τοπολογία να τρέξει και εφόσον του παρέχονται και τα high level constraints μπορεί να βρει το κατάλληλο configuration που ικανοποιεί τις ανάγκες του γρήγορα και να φτιάξει αντίστοιχο execution plan.

Μέσω αυτών των δύο μηχανισμών εξερευνήσαμε δύο διαφορετικούς τρόπους προσέγγισης και επίλυσης του ίδιου προβλήματος. Η πιο αποτελεσματική λύση θα ήταν ένας μηχανισμός που να ενώνει τις δύο προσεγγίσεις. Ένας τέτοιος μηχανισμός θα μπορούσε να χρησιμοποιήσει τον predictor για να κάνει “pivot” σε ένα αρχικό configuration και στη συνέχεια να χρησιμοποιήσει το rule based για να παρακολουθεί την πορεία του συστήματος καθ’ όλη τη διάρκεια της εκτέλεσης του. Με αυτό το τρόπο συνδυάζουμε τα πλεονεκτήματα των δύο μηχανισμών ενώ ταυτόχρονα καλύπτουμε τις αδυναμίες τους.

5 Βιβλιογραφία

[1] D. Sun, G. Zhang, W. Zheng and K. Li, "Key Technologies for Big Data Stream Computing," SUNY, New Paltz, 2014.

[2] Apache Storm Issue Tracking, "Auto-Scaling Resources in a Topology," STORM-594, https://issues.apache.org/jira/browse/STORM-594.

[3] Apache Storm, "Concepts," http://storm.apache.org/releases/1.0.0/Concepts.html.

[4] Apache Storm, "Setting up a Storm Cluster," http://storm.apache.org/releases/1.0.0/Setting-up-a-Storm-cluster.html.

[5] Apache Storm, "Daemon Fault Tolerance," http://storm.apache.org/releases/1.0.0/Daemon-Fault-Tolerance.html.

[6] M. G. Noll, "Understanding the Internal Message Buffers of Storm," http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/.

[7] Apache Storm, "Guaranteeing Message Processing," http://storm.apache.org/releases/1.0.0/Guaranteeing-message-processing.html.

[8] Apache Storm, "Understanding the Parallelism of a Storm Topology," http://storm.apache.org/releases/1.0.0/Understanding-the-parallelism-of-a-Storm-topology.html.

[9] WEKA, "Data Mining Software in Java," http://www.cs.waikato.ac.nz/ml/weka/.

[10] MOEA, "Java Framework for Multiobjective Optimization," http://moeaframework.org/.