ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Διπλωματική εργασία για το Μεταπτυχιακό Δίπλωμα Ειδίκευσης στα

<<Ολοκληρωμένα Συστήματα Υλικού και Λογισμικού>>

Τίτλος Μεταπτυχιακής Διπλωματικής Εργασίας:

«Αλγόριθμοι Εξόρυξης Διαδικασιών στο Περιβάλλον Ανάπτυξης Spark»

Βλάσης Πίτσιος
Α.Μ.: 220

Επιβλέπων Καθηγητής:
Χρήστος Ζαρολιάγκης, Καθηγητής Παν. Πατρών

Τριμελής Εξεταστική Επιτροπή:
Χρήστος Ζαρολιάγκης, Καθηγητής Παν. Πατρών
Σπυρίδων Σιούτας, Καθηγητής Παν. Πατρών
Γιάννης Τζήμας, Αναπ. Καθηγητής ΤΕΙ Δυτικής Ελλάδος

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. **Χρήστο Ζαρολιάγκη** που ανέλαβε την επίβλεψη της διπλωματικής μου εργασίας και μου έδωσε την δυνατότητα να ασχοληθώ με το περιβάλλον ανάπτυξης εφαρμογών Spark.

Επίσης, θα ήθελα να ευχαριστήσω τους κ.κ. **Σπυρίδων Σιούτα** και **Γιάννη Τζήμα** που δέχτηκαν να είναι μέλη της εξεταστικής επιτροπής για την εξέταση και κρίση της μεταπτυχιακής μου εργασίας.

Ακόμα θα ήθελα να ευχαριστήσω το διδάκτορα **Μανόλη Βιέννα**, για τη βοήθεια και τις συμβουλές του κατά την διάρκεια της εκπόνησης της μεταπτυχιακής μου εργασίας.

Σύνοψη

Η εξόρυξη διαδικασιών (process mining) αποτελεί έναν συνδετικό κρίκο ανάμεσα στην εξόρυξη δεδομένων (data mining) και στην διαχείριση διαδικασιών σε επιχειρήσεις (business process management). Συγκεκριμένα αποτελεί μια οικογένεια από τεχνικές που υποστηρίζουν την ανάλυση των διαδικασιών βασιζόμενες σε διάφορα σύνολα γεγονότων (event logs). Σκοπός της εξόρυξης διαδικασιών (process mining) είναι η κατανόηση αλλά και η βελτίωση της απόδοσης των διαδικασιών μιας επιχείρησης.

Η παρούσα εργασία επικεντρώθηκε στο κατά πόσο είναι δυνατή η παραλληλοποίηση τέτοιων αλγορίθμων εξόρυξης διαδικασιών και ποιο είναι το όφελος από την παράλληλη εκτέλεση τέτοιων αλγορίθμων. Συγκεκριμένα μελετήθηκε ο αλγόριθμος Alpha (Alpha Algorithm), ο οποίος κατασκευάζει διαγράμματα PetriNet από ακολουθίες γεγονότων. Στόχος ήταν αφενός η ανάπτυξη του συγκεκριμένου αλγορίθμου χωρίς την χρήση τεχνικών παραλληλοποίησης και αφετέρου η ανάπτυξη του ίδιου αλγορίθμου με τεχνικές mapReduce σε περιβάλλον ανάπτυξης Spark, όπου ο αλγόριθμος εκτελείται παράλληλα. Στη συνέχεια έγινε εκτέλεση πειραμάτων σε πραγματικά δεδομένα και αξιολόγηση των αποτελεσμάτων της σειριακής εκτέλεσης του αλγορίθμου αλλά και της παράλληλης εκτέλεσης σε μια συστάδα υπολογιστών (cluster) με χρήση του περιβάλλοντος ανάπτυξης Spark. Η ανάπτυξη του αλγόριθμου Alpha υλοποιήθηκε σε Scala και αξιολογήθηκε εκτενώς πειραματικά σε πραγματικά δεδομένα.

Λέξεις κλειδιά: Spark, Scala, Process mining, Databricks, Alpha Algorithm

Abstract

Process mining is a link between data mining and business process management. In particular, it is a family of techniques that support the analysis of processes based on event logs. The purpose of process mining is to understand and improve the performance of business processes.

This research proposal focuses on how parallelization of such process mining algorithms is possible and what is the benefit of the parallel execution of such algorithms. In particular, the Alpha Algorithm (a process mining algorithm), which builds PetriNet graphs from events sequences, was studied.The aim was to develop this algorithm without using parallelism techniques and then to develop the same algorithm with mapReduce techniques in Spark development environment, where the algorithm runs in parallel. In the next stage, experiments were performed on real data and the results were evaluated by comparing the serial execution of the algorithm with the parallel execution on a cluster using the Spark framework. The development of the Alpha algorithm was implemented with Scala source code and was extensively evaluated experimentally on real data.

Keywords: Spark, Scala, Process mining, Databricks, Alpha Algorithm

Περιεχόμενα

Περιεχόμενα

1. Εισαγωγή

Στο κεφάλαιο αυτό παρουσιάζεται μια εισαγωγή στην εξόρυξη δεδομένων και διαδικασιών καθώς και οι στόχοι, η συνεισφορά και η δομή της παρούσας διπλωματικής εργασίας.

1.1 Εξόρυξη δεδομένων και διαδικασιών

1.1.1 Τι είναι "εξόρυξη διαδικασιών"

Στην καθημερινή ζωή καταγράφονται πάρα πολλα γεγονότα (**events**), παραδείγματος χάριν στατιστικά από μέσα κοινωνικής δικτύωσης (facebook, twitter, linkedin), από ιστοσελίδες παραγγελιών προιόντων (ebay, amazon) κτλπ. Αυτό παρέχει μια αστείρευτη πηγή πληροφοριών που μπορούν να παρέχουν πολύτιμη γνώση χρησιμοποιώντας την εξόρυξη διαδικασιών (**process mining**). Η εξόρυξη διαδικασιών μπορεί να χρησιμοποιηθεί για να παράγει μοντέλα συμπεριφοράς (**behavioral models**) που να δείχνουν τι πραγματικά κάνουν οι άνθρωποι και οι επιχειρήσεις. Το αντικείμενο ενδιαφέροντος των δεδομένων πολύ μεγάλου όγκου (**Big Data**) είναι υπερβολικά επικεντρωμένο στα δεδομένα (**data**), στην αποθήκευση (**storage**) και στην επεξεργασία (**processing**) τους, και όχι τόσο στις διαδικασίες (**processes**) που θα ήταν χρήσιμο για μια επιχείρηση να αναλυθούν και να βελτιωθούν. Για αυτό ακριβώς το λόγο υπάρχει η εξόρυξη διαδικασιών (**process mining**) η οποία παρέχει μια καινούρια οπτική στα δεδομένα (Big or Small data) και παρέχει τα απαραίτητα εργαλεία για να ξεκινήσει η ανάλυση πραγματικών συμπεριφορών βασιζόμενοι σε πραγματικά γεγονότα (**event data**) που μπορούν να βρεθούν σε οποιαδήποτε επιχείρηση.

Η εξόρυξη διαδικασιών είναι ένα ξεχωριστό εργαλείο για κάποιον επιστήμονα δεδομένων (**data scientist**). Η εξόρυξη διαδικασιών γεφυρώνει το χάσμα που υπάρχει ανάμεσα στην ανάλυση διαδικασιών που βασίζεται σε μοντέλα (**model-based process analysis**) όπως η προσομοίωση, και των τεχνικών ανάλυσης που βασίζονται σε δεδομένα (**data-centric analysis techniques**) όπως η μηχανική μάθηση (**machine learning**) και η εξόρυξη δεδομένων.

Σύμφωνα με τα παραπάνω, μπορούμε να ορίσουμε την εξόρυξη διαδικασιών (**process mining**) ως μια τεχνική διαδικασίας διαχείρησης, που είναι χρήσιμη για την ανάλυση επιχειρηματικών διαδικασιών βασισμένες σε καταγραφή γεγονότων. Η βασική ιδέα είναι να εξορύξουμε γνώση από καταγεγραμμένα γεγονότα αποθηκευμένα σε ένα σύστημα πληροφοριών. Η εξόρυξη διεργασιών στοχεύει στο να βελτιώσει αυτό το σύστημα πληροφοριών με παρεχόμενες τεχνικές και εργαλεία για να ανακαλύψει τη διαδικασία, τον έλεγχο, τα δεδομένα και τις κοινωνικές δομές από την καταγραφή γεγονότων.

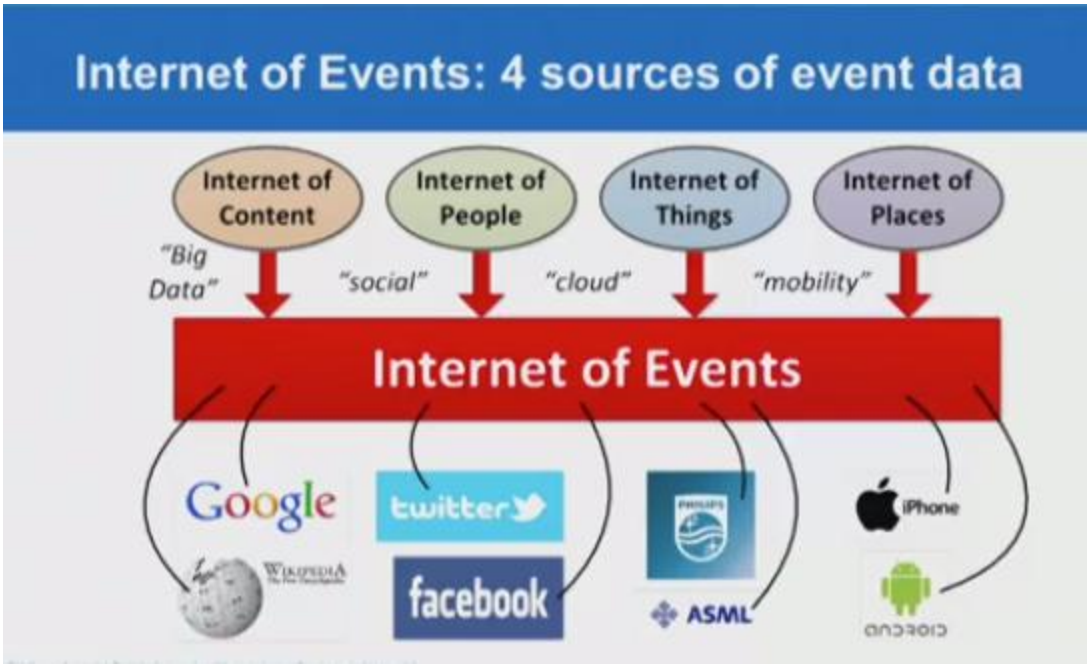
Οι τεχνικές της εξόρυξης διεργασιών συχνά χρησιμοποιούνται όταν δεν υπάρχει καμία επίσημη περιγραφή της διαδικασίας για το πώς μπορεί να αποκτηθεί με άλλους τρόπους, ή όταν η ποιότητα μιας υπαρκτής τεκμηρίωσης είναι αμφισβητήσιμη. Για παράδειγμα, οι οικονομικοί έλεγχοι ενός συστήματος ροής διαχείρησης, η διεξαγωγή καταγραφής ενός συστήματος σχεδιασμού επιχειρηματικών πόρων, και το ηλεκτρονικό σύστημα καταχώρησης ενός νοσοκομείου μπορεί να χρησιμοποιηθεί για να ανακαλύψει μοντέλα που να περιγράφουν διαδικασίες, οργανισμούς και προϊόντα. Ακόμη, αυτή η καταγραφή γεγονότων μπορεί να χρησιμοποιηθεί για να συγκρίνουμε καταγεγραμμένα γεγονότα με κάποιο προγενέστερο πρότυπο για να δούμε εάν η παρατηρούμενη πραγματικότητα συμμορφώνεται σε κάποιο κατευθυντήριο ή περιγραφικό μοντέλο.

1.1.2 Επιστήμη των δεδομένων και Μεγάλα Δεδομένα

Σήμερα, περισσότερο από ποτέ, υπάρχει διαθέσιμη τεράστια ποσότητα δεδομένων η οποία συλλέγεται από διάφορες πηγές. Αυτό μπορεί να γίνει αντιληπτό, αν σκεφτεί κανείς τον όγκο των δεδομένων που έχουν παραχθεί από τα προιστορικά χρόνια μέχρι το 2003 και το ότι αυτή η ποσότητα πλέον παράγεται μέσα σε 10 λεπτά της ώρας. Αυτό ακριβώς δείχνει την απίστευτη ανάπτυξη των δεδομένων. Αυτά τα δεδομένα παράγονται οποιαδήποτε στιγμή από οποιονδήποτε άνθρωπο την στιγμή που κάνει χρήση της πιστωτικής του κάρτας, την στιγμη που συνομιλεί στο τηλέφωνο, την στιγμή που αγοράζει τρόφιμα από το σούπερ μάρκετ.

Η πιο σημαντική όμως πηγή δεδομένων είναι τα δεδομένα που διακινούνται μέσω διαδικτύου (**internet of events**). Υπάρχουν 4 είδη τέτοιων δεδομένων

- **Internet of content:** Στην συγκεκριμένη κατηγορία ανήκουν ιστοσελίδες από το κλασικό διαδίκτυου, όπως είναι γνωστό σε όλους, δηλαδή Google και Wikipedia. Και συνήθως όταν οι άνθρωποι μιλάνε για Μεγάλα Δεδομένα αναφέρονται σε αυτό το είδος δεδομένων.
- **Internet of people:** Στην συγκεκριμένη κατηγορία ανήκουν τα μέσα κοινωνικής δικτύωσης, όπως το Twitter και το Facebook, τα οποία παράγουν τεράστιο όγκο δεδομένων.
- **Internet of things:** Στην συγκεκριμένη κατηγορία ανήκουν τα δεδομένα που παράγονται από διάφορες συσκευές που είναι συνδεδεμένες στο διαδίκτυο όπως ο εκτυπωτής, η τηλεόραση, οι παιχνιδομηχανές. Στο μέλλον όλο και περισσότερες συσκευές θα συνδέονται στο διαδίκτυο, όπως το το ψυγείο ή το πλυντήριο και αυτό θα έχει σαν αποτέλεσμα την παραγωγή ακόμα μεγαλύτερου όγκου δεδομένων.
- **Internet of places:** Όταν παραδείγματος χάριν ο χρήστης χρησιμοποιεί το κινητό του τηλέφωνο το οποίο έχει πάνω του διάφορους αισθητήρες τοποθεσίας και καταγράφει την θέση του. Αυτό είναι άλλη μια πηγή δεδομένων.



Εικόνα 1:
Οι 4 πηγές παραγωγής δεδομένων

Εύκολα λοιπόν γίνεται κατανοητό το πως τα δεδομένα έχουν αποκτήσει τεράστια σημασία τα τελευταία χρόνια αλλά και γιατί ο κόσμος ασχολείται όλο και περισσότερο μαζί τους. Αλλά η πρόκληση σήμερα δεν είναι η παραγωγή περισσότερων δεδομένων αλλά η μετατροπή των δεδομένων αυτών σε δεδομένα με πραγματική αξία.

Τα Μεγάλα Δεδομένα (**Big Data**) έχουν τα ακόλουθα τέσσερα(4) χαρακτηριστικά

- **Volume** (όγκος δεδομένων)
- **Velocity** (ταχύτητα): Η ταχύτητα είναι μια μεγάλη πρόκληση για τους μηχανικούς που επεξεργάζονται τα δεδομένα καθώς δεν είναι μόνο τεράστιος ο όγκος των δεδομένων που αποτελεί πρόκληση αλλά και το ότι παράγονται με μεγάλο ρυθμό και αλλάζουν πολύ γρήγορα.
- **Variety** (ποικιλία): Δεν υπάρχει μόνο ένα είδος δεδομένων, αλλά πάρα πολλά διαφορετικά είδη τα οποία ποικίλουν από κείμενο σε εικόνες αλλά και πολλά άλλα είδη. Έτσι είναι μεγάλη η πρόκληση του να συνδυαστούν όλα αυτά τα δεδομένα από εντελώς διαφορετικές πηγές.
- **Veracity** (εγκυρότητα): Αυτό σημαίνει ότι ποτέ δεν μπορούμε να είμαστε σίγουροι για την ακρίβεια των δεδομένων που έχουν καταγραφεί. Ένα παράδειγμα είναι ότι ποτέ δεν μπορούμε να είμαστε σίγουροι για το ποιος είναι ο χρήστης της συσκευής που παράγει τα δεδομένα καθώς αυτά μπορεί να διαφοροποιούνται από χρήστη σε χρήστη.

Λόγω λοιπόν των παραπάνω, τα τελευταία χρόνια έχει δημιουργηθεί ένα νέο επάγγελμα στην επιστήμη των υπολογιστών, αυτό του επιστήμονα δεδομένων (**data scientist**). Οι στόχοι ενός επιστήμονα δεδομένων είναι να συλλέξει, να αναλύσει αλλά και να μεταφράσει τα δεδομένα από πολλές διαφορετικές πηγές αλλά και να τα επεξεργαστεί με τέτοιο τρόπο ώστε να εξάγει χρήσιμα αποτελέσματα. Είναι βέβαιο ότι τα επόμενα χρόνια λόγω της τεράστιας ανάπτυξης των δεδομένων το συγκεκριμένο επάγγελμα θα αποκτήσει όλο και μεγαλύτερη αξία.

Στόχος ενός επιστήμονα δεδομένων είναι να απαντήσει τις παρακάτω ερωτήσεις που σχετίζονται με κάποια γεγονότα (**events**)

- Τι συνέβει; Αν παράδειγμα εμφανιστούν κάποια σημεία συμφόρησης (**bottlenecks**) ή κάποιες αποκλίσεις (**deviations**) στα καταγεγραμμένα γεγονότα (**events**) τότε αυτό σημαίνει ότι κάτι μη προβλεπόμενο προέκυψε στην διαδικασία.
- Η δεύτερη ερώτηση που προκύπτει είναι γιατί προέκυψαν τα παραπάνω. Που υπήρξε η καθυστέρηση; Γιατί τα γεγονότα ακολούθησαν ένα μη αναμενόμενο μονοπάτι;
- Οι δυο παραπάνω ερωτήσεις αφορούν το παρελθόν. Αλλά φυσικά ο επιστήμονας δεδομένων πρέπει να απαντήσει ερωτήσεις για το μέλλον. Άρα η τρίτη ερώτηση είναι τι θα συμβεί στο μέλλον και τι μπορούμε να προβλέψουμε από τις πληροφορίες που έχουμε ώστε να κάνουμε προβλέψεις για το τι θα συμβεί.
- Η τέταρτη ερώτηση της επιστήμης των δεδομένων είναι να αναρωτηθεί κάποιος ποιο είναι το καλύτερο επιθυμητό αποτέλεσμα που μπορεί να συμβεί σε μια διαδικασία. Άρα λοιπόν, ο επιστήμονας δεδομένων πρέπει να προτείνει συγκεκριμένα πράγματα ώστε να βελτιωθεί η τρέχουσα καταστασή και να αφαιρεθούν τα σημεία συμφόρησης και τα γεγονότα να μην αποκλίνουν από την επιθυμητή συμπεριφορά που θα έχει σαν επακόλουθο την παροχή μιας καλύτερης υπηρεσίας.

Για παράδειγμα, μπορούμε να αναρωτηθούμε για τις 4 αυτές ερωτήσεις στις ροές παροχής διαδικασιών φροντίδας σε ένα νοσοκομείο. Θα ήταν δυνατό να ερωτηθούν οι παρακάτω ερωτήσεις

- Γιατί οι ασθενείς περιμένουν τόσο πολύ;
- Τι προκαλεί αυτές τις καθυστερήσεις;
- Ακολουθούν οι γιατροί τις κατευθυντήριες γραμμές του εκάστοτε νοσοκομείου;
- Μπορούμε να προβλέψουμε τις καθυστερήσεις;
- Πόσο προσωπικό θα χρειαστεί τις επόμενες μέρες ένα νοσοκομείο;
- Πως μπορούν να μειωθούν τα κόστη χωρίς να μειωθεί η ποιότητα;

Έχει γίνει αρκετή ανάλυση δεδομένων στα νοσοκομεία και έχει διαπιστωθεί ότι διαφορετικοί γιατροί επεξεργάζονται παρόμοια πράγματα με εντελώς διαφορετικό τρόπο και αυτός είναι ένας λόγος για τις μεγάλες αναμονές των ασθενών. Αυτές λοιπόν οι ερωτήσεις ως ένα σημείο μπορούν να οριστούν σαν μια επιχειρηματική διαδικασία (**business process**).

Αλλά ακόμα και αν ψάξουμε για δεδομένα σε μια συσκευή, όπως μια συσκευή ακτίνων Χ, θα δούμε ότι παράγεται ένας πολύ μεγάλος αριθμός δεδομένων και μπορούν να ερωτηθούν διάφορες ερωτήσεις που όλες μπορούν να απαντηθούν αναλύοντας τα δεδομένα προσεκτικά. Ορισμένα παραδείγματα ερωτήσεων

- Πως ακριβώς χρησιμοποιούνται αυτές οι συσκευές στα νοσοκομεία;
- Πότε αυτές οι συσκευές ξεκινάνε να παρουσιάζουν προβλήματα;
- Πότε καταστρέφονται εντελώς και δεν είναι πλέον λειτουργικές;
- Ποια εξαρτήματα πρέπει να αντικατασταθούν;
- Όταν η συσκευή σταματήσει να λειτουργεί μπορούμε να παράγουμε διαγνωστικές πληροφορίες που να υποδεικνύουν ποια εξαρτήματα έχουν καταστραφεί, ώστε να είναι δυνατή η άμεση επισκευή από έναν μηχανικό;
- Μπορούμε να προβλέψουμε πότε μια συσκευή θα έχει βλάβη;
- Μπορούμε να μάθουμε από τα υπάρχοντα προβλήματα ποια εξαρτήματα χρειάζονται βελτίωση ώστε μια συσκευή να έχει μεγαλύτερη διάρκεια λειτουργείας χωρίς προβλήματα;

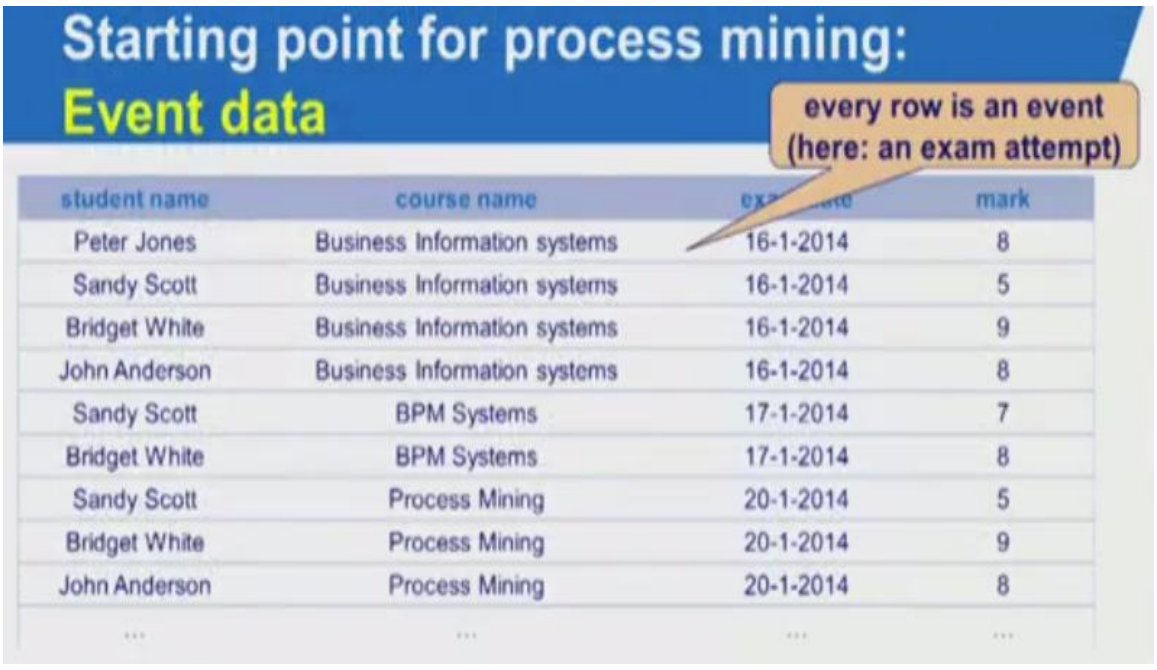
Όπως γίνεται λοιπόν αντιληπτό, κάποιος επιστήμονας δεδομένων χρειάζεται διαφορετικές δεξιότητες για να απαντήσει όλες αυτές τις ερωτήσεις. Κάποιες προφανείς δεξιότητες είναι η στατιστική (**statistics**) και η εξόρυξη δεδομένων. Ακόμα θα πρέπει να μπορεί να χειριστεί απίστευτα μεγάλες βάσεις δεδομένων αλλά και να έχει γνώση κοινωνικών επιστημών για να μπορεί να αντιλαμβάνεται έννοιες όπως η δεοντολογία (**ethics**) και η ιδιωτικότητα (**privacy**). Το πιο σηματικό όμως είναι να μπορεί να εφαρμόσει τεχνικές εξόρυξης διαδικασιών (**process mining techniques**) πάνω στα δεδομένα ώστε να μπορεί να βελτιώσει αλλά και να μάθει τις διαδικασίες μέσα σε μια επιχείρηση ή έναν οργανισμό.

Με λίγα λόγια μπορούμε να πούμε ότι η εξόρυξη διαδικασιών είναι μια υπο-κατηγορία της επιστήμης των δεδομένων, αλλά στοχεύει στην ανάλυση των διαδικασιών βασιζόμενο στα δεδομένα και το κλειδί είναι οι διαδικασίες (**processes**). Η εξόρυξη διαδικασιών διαφέρει από την εξόρυξη δεδομένων διότι δεν ενδιαφερόμαστε να απομονώσουμε αποφάσεις ή πρότυπα χαμηλού επιπέδου (low level patterns). Ο βασικός στόχος της εξόρυξης διαδικασιών είναι η βελτιστοποίηση των από άκρη σε άκρη (**end-to-end**) διαδικασιών σύμφωνα με την αλληλεπίδραση ανάμεσα στα δεδομένα γεγονότων, στις διαδικασίες και στα μοντέλα διαδικασιών (**process models**). Όλα αυτά μπορούν να αναφερθούν και σαν πληροφορία διαδικασιών (**process intelligence**).

1.1.3 Διάφορα είδη εξόρυξης διαδικασιών

Από το προηγούμενο κεφάλαιο γίνεται κατανοητό ότι έχουμε πολύ μεγάλο μέγεθος δεδομένων, αλλά η εξόρυξη διαδικασιών δεν έχει να κάνει με την συλλογή δεδομένων αλλά με την ανάλυση διαδικασιών. Η εξόρυξη διαδικασιών γεφυρώνει το κενό ανάμεσα στην κλασική ανάλυση διαδικασιών μέσω μοντέλων (**process model analysis**) και της ανάλυσης η οποία βασίζεται στα δεδομένα όπως το εξόρυξη δεδομένων και η μηχανική μάθηση. Αυτό συμβαίνει γιατί η εξόρυξη διαδικασιών επικεντρώνεται στις διαδικασίες χρησιμοποιώντας ταυτόχρονα πραγματικά δεδομένα. Στη κλασική εξόρυξη διαδικασιών, οι επιστήμονες δεδομένων συνήθως δεν επεξεργάζονται τις διαδικασίες και ειδικά τις από άκρη σε άκρη διαδικασίες. Αντίθετα στις περιοχές που αφορούν την ανάλυση διαδικασιών μέσω μοντέλων, συνήθως τα δεδομένα αγνοούνται. Για αυτό το λόγο η εξόρυξη διαδικασιών είναι πολύ σημαντική επειδή απαντάει ερωτήσεις σχετιζόμενες με την απόδοση (**performance**) των διαδικασιών αλλά και με την συμμόρφωση (**compliance**) των δεδομένων πάνω σε μια συγκεκριμένη διαδικασία.

Το αρχικό σημείο από το οποίο θα ξεκινήσει κάποιος την εξόρυξη διαδικασιών είναι τα δεδομένα γεγονότων. Στην παρακάτω εικόνα βλέπουμε έναν πίνακα στον οποίο κάθε γραμμή αναφέρεται σε ένα δεδομένο (**event)**.



student name	course name	exam date	mark
Peter Jones	Business Information systems	16-1-2014	8
Sandy Scott	Business Information systems	16-1-2014	5
Bridget White	Business Information systems	16-1-2014	9
John Anderson	Business Information systems	16-1-2014	8
Sandy Scott	BPM Systems	17-1-2014	7
Bridget White	BPM Systems	17-1-2014	8
Sandy Scott	Process Mining	20-1-2014	5
Bridget White	Process Mining	20-1-2014	9
John Anderson	Process Mining	20-1-2014	8
...	...	...	...

Εικόνα 2:
Αλληλουχία γεγονότων δεδομένων

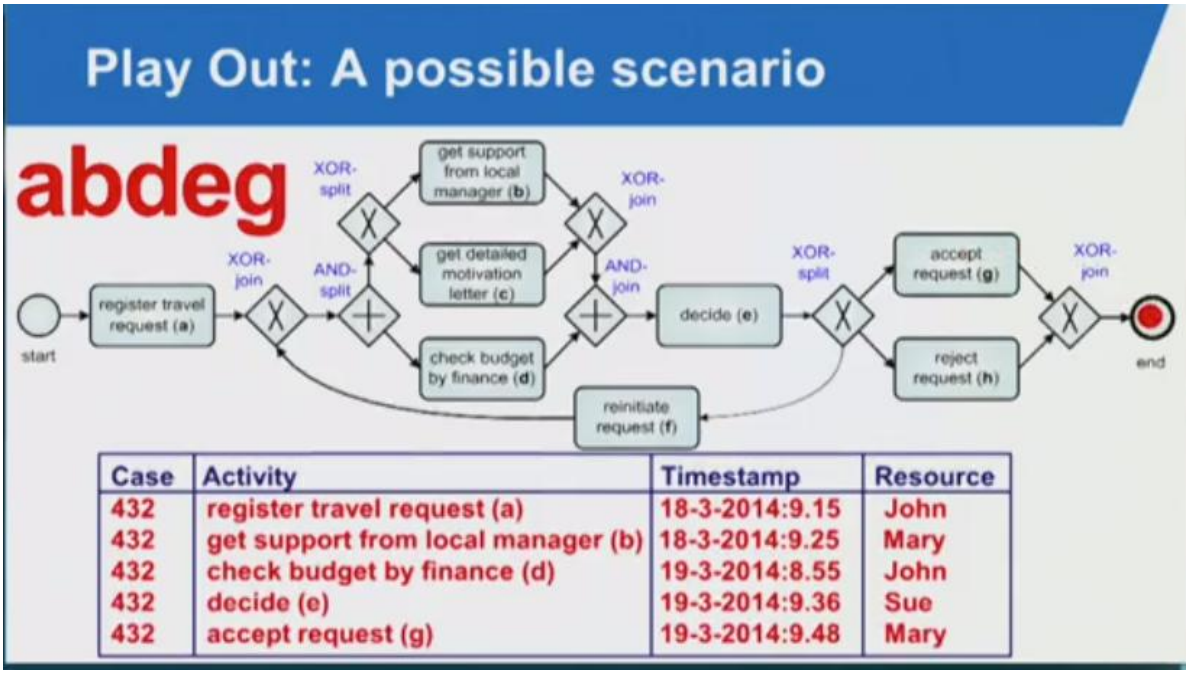
Ένα δεδομένο έχει συγκεκριμένες ιδιότητες (**properties**) και για αυτό υπάρχουν διαφορετικοί τύποι ιδιοτήτων.

- Η πρώτο ιδιότητα είναι το **caseId**. Στην συγκεκριμένη περίπτωση κάθε εγγραφή του πίνακα αναφέρεται σε έναν μαθητή ο οποίος δίνει μια εξέταση σε κάποιο μάθημα. Άρα το **caseId** αναφέρεται στον μαθητή.
- Η δεύτερη ιδιότητα είναι το όνομα μιας δραστηριότητας (**activity name**) και αναφέρεται στην εξέταση.
- Η τρίτη ιδιότητα είναι η χρονική στιγμή (**timestamp**) που στην συγκεκριμένη περίπτωση είναι η ημερομηνία της εξέτασης.

Ακόμα θα μπορούσαν να υπάρχουν και άλλα δεδομένα στο παράδειγμα όπως η βαθμολογία, αλλά οι παραπάνω τρεις ιδιότητες είναι τα απολύτως απαραίτητα για μια ανάλυση της εξόρυξης διαδικασιών.

Κύριως στόχος της εξόρυξης διαδικασιών είναι η σχέση ανάμεσα στα μοντέλα διαδικασιών (**process models**) και τα γεγονότα (**event data**). Υπάρχουν τρία είδη σχέσεων ανάμεσα σε μοντέλα διαδικασιών και τα γεγονότα**.** Τα οποία είναι τα **Play-out**, **Play-in** και **Replay**.

Στο **Play-out** η βασική ιδέα είναι ότι ξεκινάς από ένα μοντέλο και από αυτό το μοντέλο παράγεις την συμπεριφορά της διαδικασίας. Αυτό φαίνεται ξεκάθαρα στην παρακάτω εικόνα όπου έχουμε ένα μοντέλο **BPMN**.

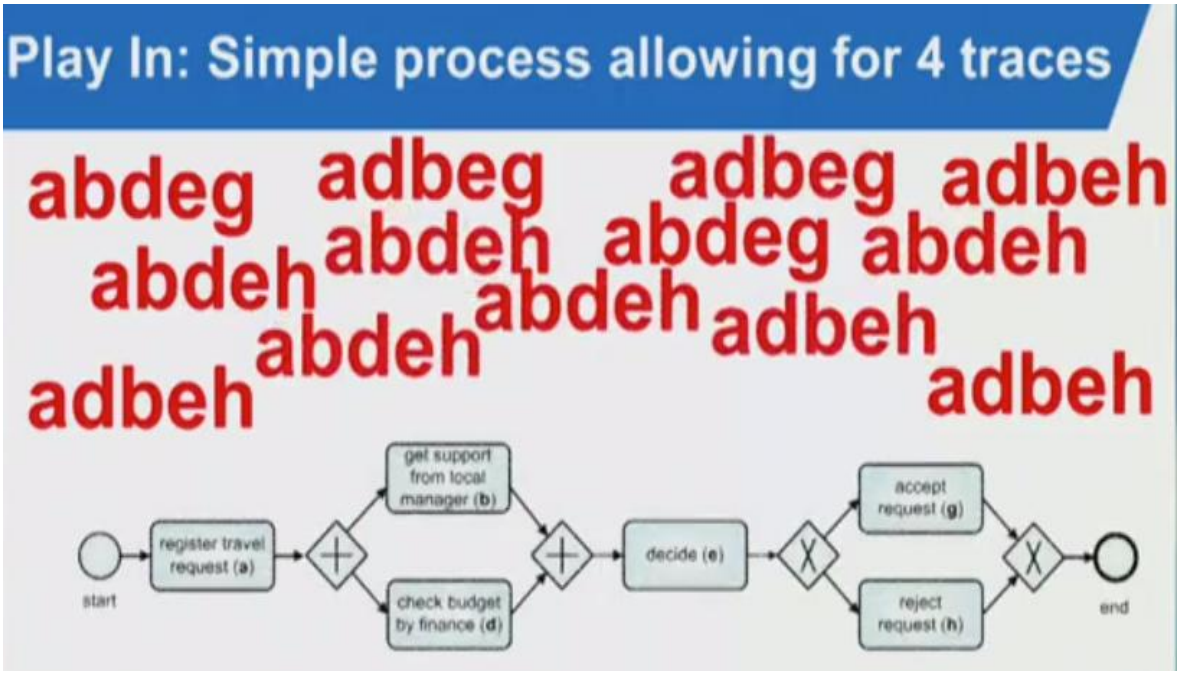


Εικόνα 3:
Play-out. Από ένα BPMN μοντέλο κατασκευάζεται ένα γεγονός (abdeg)

Το συγκεκριμένο BPMN μοντέλο έχει να κάνει με τον χειρισμό αιτήσεων ταξιδιού. Πάντα ξεκινάει από κάποια δραστηριότητα **a** όπου καταχωρείται το αίτημα ταξιδιού. Συμβαίνει σε

μια συγκεκριμένη χρονική στιγμή και εκτελείται από ένα συγκεκριμένο άτομο. Παρατηρούμε ότι στο μοντέλο αυτό παράγεται το γεγονός abdeg με τελικό στάδιο την δραστηριότητα **g**. Το μοντέλο αυτό βέβαια μπορεί να παράξει ένα μεγάλο συνδιασμό από γεγονότα ανάλογα με τις επιλογές που κάνει ο κάθε χρήστης που το εκτελεί. Το γεγονός abdeg είναι μόνο ένα από αυτά. Με λίγα λόγια τα γεγονότα αυτά αποτελούν πιθανούς τρόπους να γίνει playing out το μοντέλο. Αν κάποιος χρήστης χρησιμοποιεί εξομοίωση ή χτίζει κάποιο πληροφοριακό σύστημα το οποίο οδηγείται από τέτοια μοντέλα, τότε το play-out είναι αυτό ακριβώς που κάνει.

Στην ακριβώς αντίστροφη περίπτωση (**Play-In**), ξεκινάμε από τα δεδομένα γεγονότων προσπαθώντας να κατασκευάσουμε το αντίστοιχο μοντέλο διαδικασιών. Φυσικά υπάρχουν πολλοί αλγόριθμοι που μπορούν να επιτύγχουν την δημιουργία ενός μοντέλου διαδικασιών από δεδομένα γεγονότων. Ένας από αυτούς είναι και ο αλγόριθμος **Alpha**, η μελέτη του οποίου είναι και αντικείμενο της παρούσας εργασίας. Η βασική ιδέα είναι ότι κοιτώντας έναν αριθμό από παραδείγματα συμπεριφοράς, δηλαδή γεγονότα, αυτόματα παράγεται ένα μοντέλο από αυτά. Αυτό φαίνεται στην παρακάτω εικόνα.



Εικόνα 4:
Play-in. Από έναν αριθμό από γεγονότα κατασκευάζεται ένα μοντέλο BPMN.

Στην εικόνα 4 φαίνεται η διαδικασία του **Play-In**. Αν παρατηρήσουμε προσεκτικά τα γεγονότα ή ίχνη από δραστηριότητες (**traces)** που φαίνονται στο πάνω μέρος της εικόνας, βλέπουμε ότι όλα τα γεγονότα ξεκινούν με **a**. Άρα συμπεραίνουμε ότι και το εξαγώμενο μοντέλο θα ξεκινάει και αυτό με **a**. Όλα τα γεγονότα καταλήγουν σε **g** ή **h**. Άρα και στο

εξαγώμενο μοντέλο στο τέλος υπάρχει μια επιλογή για να γίνει αποδεκτή ή όχι με μια δραστηριότητα **g** ή **h**. Φυσικά με ένα πιο πολύπλοκο σύνολο γεγονότων**,** τόσο πιο πολύπλοκο θα γίνει και το παραγώμενο μοντέλο. Το σημαντικότερο που πρέπει να γίνει κατανοητό σε αυτό το σημείο είναι ότι δεν υπάρχει κάποιο αρχικό μοντέλο, αλλά παράγεται αυτόματα από την επεξεργασία των αρχικών γεγονότων.

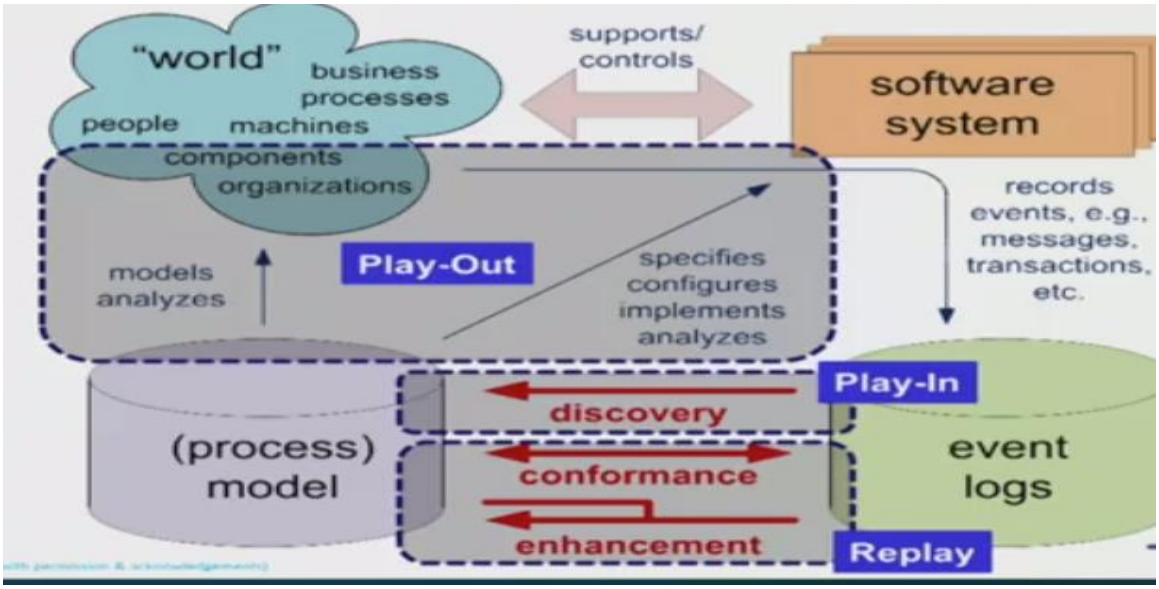
Το τρίτο είδος σχέσεων ανάμεσα σε μοντέλα διαδικασιών και δεδομένα γεγονότων λέγεται **Replay**. Είναι ένα πολύ βασικό χαρακτηριστικό της εξόρυξη διαδικασιών επειδή επιτρέπει να ξανά επαναληφθεί η πραγματικότητα πάνω σε ένα μοντέλο. Δεν έχει σημασία πως παρήχθησαν τα δεδομένα είτε με το χέρι είτε μέσω κάποιας διαδικασίας που ανακαλύπτει μοντέλα (**process discovery**), το σημαντικό είναι ότι μπορεί να ξανα γίνει επανάληψη της πραγματικότητας πάνω στο μοντέλο. Κατά την διάρκεια του **Replay** ενός γεγονότος μπορεί να ταιριάζει το γεγονός που δοκιμάζουμε στο μοντέλο αλλά υπάρχει και η πιθανότητα να μην ταιριάζει. Αυτή η τεχνική ονομάζεται συμμόρφωση (**conformance checking**).

Το **Replay** δεν έχει να κάνει όμως μόνο με την συμμόρφωση αλλά και με την ανάλυση της απόδοσης (**performance analysis**) του μοντέλου. Μπορούμε πολύ εύκολα να ξανά τρέξουμε διάφορα event που έχουν καταγραφεί και έχουν πάνω τους χρονική πληροφορία (**timestamp**). Έτσι εκτελώντας τα γεγονότα πάνω στο μοντέλο στο τέλος γνωρίζουμε πόσος χρόνος δαπανήθηκε σε όλα τα διαφορετικά σημεία της διαδικασίας. Έτσι μπορούμε να δούμε πόσο χρόνο πήραν οι διάφορες δραστηριότητες αλλά και τις καθυστερήσεις μεταξύ των δραστηριοτήτων. Αυτές οι μετρήσεις μπορούν να γίνουν για δεκάδες χιλιάδες περιπτώσεις και έτσι να παραχθεί μια πιθανοτική κατανομή για αυτές.

Συνοψίζοντας τα παραπάνω, υποθέτουμε ότι έχουμε ένα σύστημα (**software system**) το οποίο με κάποιο τρόπο καταγράφει γεγονότα τα οποία συμβαίνουν. Τέτοια παραδείγματα είναι οι κάτοικοι μιας πόλης που κάνουν ένσταση στον τρόπο αξιολόγησης των σπιτιών τους θεωρώντας ότι οι φόροι που καταβάλουν είναι υψηλοί, οι ασθενείς σε ένα νοσοκομείο για τον τρόπο που εξυπηρετούνται, οι πελάτες σε ένα εστιατόριο για την ποιότητα των γευμάτων κτλπ. Με αυτό τον τρόπο συγκεκτρώνονται τα γεγονότα (**events**). Από αυτά τα γεγονότα μπορεί να γίνει η ανακάλυψη ενός μοντέλου διαδικασιών και έπειτα να γίνει συμμόρφωση πάνω στο παραγόμενο μοντέλο. Συγκρίνοντας τα γεγονότα με το μοντέλο διαδικασιών μπορούμε να εμπλουτίσουμε το μοντέλο με πληροφορία σχετική με τις αποκλίσεις (**deviations**) και την απόδοση (**performance**).

Οπότε σύμφωνα με τα παραπάνω, το **Play-out** είναι η κλασική χρήση των μοντέλων διαδικασιών, τα οποία δεν περιλαμβάνουν κάποια δεδομένα γεγονότων. Το **Play-in** αναφέρεται στην ανακάλυψη (**discovery**) ενός μοντέλου από γεγονότα. Μπορείς αυτόματα να μάθεις ένα μοντέλο διαδικασιών χωρίς καμία μοντελοποίηση από απλά δεδομένα (**raw event data**) χρησιμοποιώντας κάποιον αλγόριθμο όπως ο αλγόριθμος **Alpha**. Και φυσικά υπάρχει η δυνατότητα για το χρήστη να ξανά κάνει **Replay** τα δεδομένα για να συγκρίνει ένα μοντέλο διαδικασιών με ένα σύνολο γεγονότων, ώστε να ελέγξει και να εκτελέσει έλεγχο συμμόρφωσης πάνω στο μοντέλο.

Αυτή η διαδικασία φαίνεται στην παρακάτω εικόνα



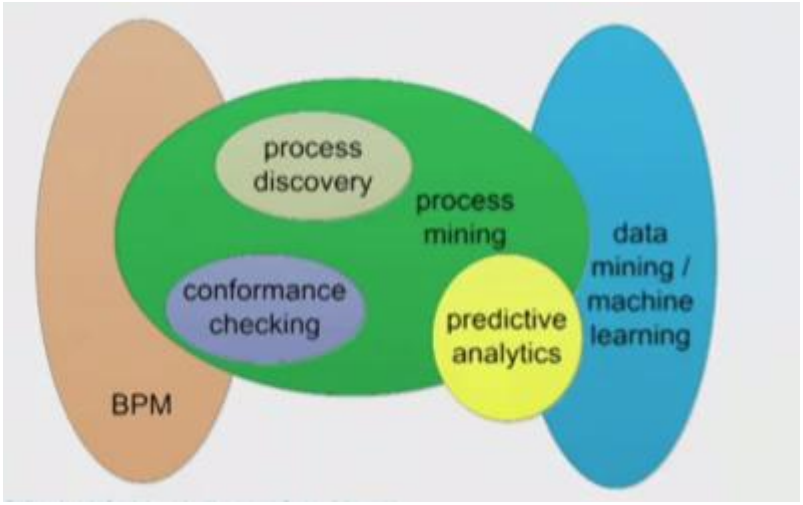
Εικόνα 5:
Σύνοψη των τριών ειδών εξόρυξης διαδικασιών

Συνοψίζοντας λοιπόν έχουμε τρεις κατηγορίες τεχνικών εξόρυξης διαδικασίων. Η ταξινόμηση αυτή βασίζεται στο κατά πόσο υπάρχει ένα προγενέστερο μοντέλο και, αν ναι, πώς χρησιμοποιείται.

- ***Ανακάλυψη (discovery)***: Δεν υπάρχει ένα προγενέστερο πρότυπο, δηλαδή, ένα έτοιμο μοντέλο διαδικασιών. Για παράδειγμα, χρησιμοποιώντας τον αλγόριθμο **Alpha**, ένα μοντέλο διαδικασίας μπορεί να ανακαλυφθεί βασιζόμενο σε χαμηλού επιπέδου γεγονότα. Υπάρχουν πολλές τεχνικές για να κατασκευάσουν αυτόματα μοντέλα διαδικασιών (για παράδειγμα δίκτυα Petri ή μοντέλα BPMN), με βάση ορισμένες καταγραφές γεγονότων. Πρόσφατα, η ερευνητική διαδικασία εξόρυξης άρχισε επίσης να στοχεύει σε άλλες προοπτικές (π.χ. τα δεδομένα, τους πόρους, το χρόνο, κλπ.).
- ***Συμμόρφωση (conformance checking)***: Υπάρχει ένα προγενέστερο μοντέλο διαδικασιών. Στόχος είναι να βρεθεί κατά πόσο διαφέρει η πραγματικότητα από το υπάρχων μοντέλο διαδικασιών. Αυτό το μοντέλο συσχετίζεται με το αρχείο καταγραφής γεγονότων και οι διαφορές μεταξύ των γεγονότων και του μοντέλου αναλύονται. Για παράδειγμα, μπορεί να υπάρχει ένα μοντέλο της διαδικασίας που αναφέρει ότι για εντολές αγοράς άνω του 1 εκατομμυρίου ευρώ, απαιτούνται δύο έλεγχοι για να ολοκληρωθεί με επιτυχία η αγορά. Συμμόρφωση μπορεί να χρησιμοποιηθεί για την ανίχνευση αποκλίσεων ώστε να εμπλουτιστεί το μοντέλο.
- ***Επέκταση (extension)***: Υπάρχει ένα μοντέλο εκ των προτέρων. Το μοντέλο αυτό επεκτείνεται με στόχο όχι να ελέγξει τη συμμόρφωση, αλλά για να εμπλουτίσει το μοντέλο.

1.1.4 Συσχέτηση εξόρυξης διαδικασιών με εξόρυξη δεδομένων

Η εξόρυξη διαδικασιών είναι ο συνδετικός κρίκος ανάμεσα ανάμεσα στην ανάλυση βασιζόμενη σε μοντέλα (**model based analysis**) και στην ανάλυση που επικεντρώνεται στα δεδομένα (**data oriented analysis**) όπως η εξόρυξη δεδομένων (**data mining**). Με την εξόρυξη διαδικασιών μπορούμε να απαντήσουμε ερωτήματα σχετικά με την απόδοση (**performance**) των διαδικασιών αλλά και ερωτήματα τα οποία σχετίζονται με την συμμόρφωση (**compliance**) πάνω σε κάποια μοντέλο. Άρα θα μπορούσαμε να πούμε ότι η εξόρυξη διαδικασιών είναι η «κόλλα» ανάμεσα στα δεδομένα και στις διαδικασίες, ανάμεσα στους εργαζομένους της τεχνολογίας της πληροφορίας αλλά και της διαχείρησης επιχειρήσεων αλλά και ανάμεσα στην απόδοση και την συμμόρφωση. Και φυσικά η εξόρυξη διαδικασιών μπορεί να γίνει και κατά την διάρκεια της εκτέλεσης αλλά και κατά την διάρκεια του σχεδιασμού. Είναι ο συνδετικός κρίκος ανάμεσα σε πολλά πράγματα και μπορεί να φανεί απίστευτα πολύτιμο.



Εικόνα 6:
Σύνδεση διαχείρησης επιχειρηματικών διαδικασιών **(BPM)** με την κλασική ανάλυση δεδομένων όπως η εξόρυξη δεδομένων**.**

Η παραπάνω εικόνα απεικονίζει πως η εξόρυξη διαδικασιών είναι ο συνδετικός κρίκος ανάμεσα στην διαχείρηση επιχειρηματικών διαδικασιών **(BPM)** και στις κλασικές κατηγορίες ανάλυσης δεδομένων όπως η εξόρυξη δεδομένων. Στην εικόνα αυτή φαίνεται ξεκάθαρα και το ότι η εξόρυξη διαδικασιών αποτελείται από άλλα είδη εξόρυξης όπως η ανακάλυψη διαδικασιών (**process discovery**) αλλά και ο έλεγχος συμμόρφωσης

(**compliance checking**). Ακόμα στην εικόνα εμφανίζεται και η ανάλυση προβλέψεων (**predictive analysis**) που σχετίζεται με την πρόβλεψη γεγονότων στο μέλλον.

Κάποιος μπορεί να σκεφτεί την εξόρυξη διαδικασιών σαν μια διαδικασία εύρεσης επιθυμητών γραμμών. Αυτές οι γραμμές δείχνουν τι συνήθως κάνουν οι άνθρωποι. Παράδειγμα ένα μονοπάτι που το ακολουθούν οι άνρθωποι με κάποιες ταμπέλες που δείχνουν που δεν πρέπει να πατήσουν. Άρα αυτές οι γραμμές μπορούν να παρομοιαστούν σαν δεδομένα γεγονότων και οι ταμπέλες σαν μοντέλα διαδικασιών. Αυτός ο παραλληλισμός δείχνει τι προσπαθεί να κάνει η εξόρυξη διαδικασιών δηλαδή να ανακαλύψει επιθυμητές γραμμές. Φυσικά πολλές φορές η διαδικασία της εξόρυξης διαδικασιών μπορεί να ανακαλύψει μη-αναμενόμενα γεγονότα που να δείχνουν ότι πολλές φορές οι άνθρωποι δεν ακολουθούν το μοντέλο διαδικασιών, δηλαδή τις επιθυμητές γραμμές.

Αρχικά, ο όρος εξόρυξη δεδομένων (**data mining**) είχε αρκετά αρνητική έννοια καθώς παρέπεμπε σε όρους όπως data snooping, fishing όπου κακόβουλοι χρήστες προσπαθούν να υποκλέψουν ευαίσθητα δεδομένα χρήστων όπως username, password, αιρθμούς πιστωτικών καρτών κτλπ. Αλλά πλέον λόγω της τεράστιας αύξησης των δεδομένων αυτό έχει αλλάξει καθώς όλο και περισσότεροι οργανισμοί ενδιαφέρονται για την οργάνωση και την ανάλυση των δεδομένων τους.

Το κοινό στοιχείο ανάμεσα στην εξόρυξη δομένων και στην εξόρυξη διαδικασιών είναι ότι η υλοποίηση τους ξεκινάει από τα δεδομένα, αλλά κατά τα άλλα υπάρχουν πολλές διαφορές ανάμεσα τους. Για τις διαδικασίες της εξόρυξης δεδομένων, οι γραμμές και οι στήλες στους πίνακες με τα δεδομένα μπορούν να σημαίνουν οτιδήποτε. Αντίθετα, στις τεχνικές της εξόρυξης διαδικασιών, υποθέτουμε ότι τα δεδομένα έχουν συγκεκριμένη μορφή και αναφέρονται σε δραστηριότητες (**activities)** σε συγκεκριμένες χρονικές στιγμές. Επιπλέον τα γεγονότα (**events**) είναι ταξινομημένα και για αυτό το λόγο ενδιαφερόμαστε σε διαδικασίες από άκρη σε άκρη (**end-to-end**).

Η βασική διαφορά τους είναι ότι η εξόρυξη δομένων έχει σαν κέντρο τα δεδομένα και όχι τις διαδικασίες. Έτσι γίνεται κατανοητό ότι η ανακάλυψη διαδιακασιών (**process discovery**) , η συμμόρφωση σε ένα μοντέλο (**conformance checking**) και η ανάλυση για την εύρεση κάποιων σημείων όπου καθυστερεί η ομαλή λειτουργεία ενός οργανισμού (**bottlenecks discovery**) δεν αποτελεί τμήμα της εξόρυξης δεδομένων αλλά τμήμα της εξόρυξης διαδικασιών. Και φυσικά τίποτα από αυτά δεν μπορεί να επιτευχθεί χρησιμοποιώντας την εξόρυξη δομένων.

Η εξόρυξη διαδικασιών συνδιάζει μοντέλα διαδικασιών και δεδομένα γεγονότων με πολλούς διαφορετικούς καινοτόμους τρόπους. Σαν αποτέλεσμα είναι ότι κάποιος χρησιμοποιώντας την εξόρυξη διαδικασιών μπορεί να ανακαλύψει τι κάνουν στην πραγματικότητα οι άνθρωποι και οι οργανισμοί. Για παράδειγμα, μοντέλα διαδικασιών μπορούν να ανακαλυφθούν αυτόματα από δεδομένα γεγονότων με την διαδικασία

εύρεσης των μοντέλων διαδικασιών. Η συμμόρφωση δεδομένων μπορεί να ελεγχθεί συγκρίνοντας μοντέλα με δεδομένα γεγονότων. Διάφορα προβλήματα σε διαδικασίες μπορούν να ανακαλυφθούν ξανά εφαρμόζοντας διάφορα γεγονότα πάνω σε μοντέλα που έχουν ήδη βρεθεί. Έτσι η εξόρυξη διαδικασιών μπορεί να χρησιμοποιηθεί για την ανακάλυψη και κατανόηση αποκλίσεων, ρίσκων αλλά και προβλήματων στην καθημερινή λειτουργία μιας εταιρείας ή ενός οργανισμού.

Τα από άκρη σε άκρη (**end-to-end)** μοντέλα διαδικασιών είναι πάρα πολύ σημαντικά και για την ανακάλυψη τους είναι απαραίτητο να εξεταστούν διαδικασίες που εκτελούνται παράλληλα (**concurrency**). Ακόμα η εξόρυξη διαδικασιών προϋποθέτει τα δεδομένα που θα αναλυθούν να έχουν μια συγκεκριμένη δομή. Τα δεδομένα αυτά ονομάζονται γεγονότα (**events**) όπως είδαμε σε προηγούμενα κεφάλαια και πρέπει να φέρουν οπωσδήποτε μια χρονική πληροφορία (**timestamp),** ένα **όνομα γεγονότος (eventName)** αλλά και κάποια μοναδικά στοιχεία που να αναφέρονται σε διάφορες περιπτώσεις (**cases**). Αυτή είναι μια σημαντική διαφορά σε σχέση με την μορφή που έχουν τα δεδομένα τα οποία θα αναλυθούν από κάποιο αλγόριθμο εξόρυξης δεδομένων.

Συνήθως όμως σε πολύ σύνθετα προβλήματα η εξόρυξη διαδικασιών και η εξόρυξη δεδομένων συνδυάζονται καθώς αποτελούν συμπληρωματικές προσεγγίσεις που η μια δυναμώνει την άλλη.

1.2 Στόχος της διπλωματικής εργασίας

Στόχος της παρούσας διπλωματικής εργασίας είναι η μελέτη και η κατανόηση της εξόρυξης διαδικασιών αλλά και η υλοποίηση του αλγόριθμου Alpha, ο οποίος είναι ένας τέτοιος αλγόριθμος. Συγκεκριμένα πρέπει να μελετηθεί το κατά πόσο είναι δυνατή η παραλληλοποίηση του αλγόριθμου Alpha και ποιο είναι το όφελος από την παράλληλη εκτέλεση αυτού του αλγορίθμου. Θα πρέπει να υλοποιηθεί ο συγκεκριμένος αλγόριθμος σε γλώσσα προγραμματισμού Scala χώρις την χρήση του περιβάλλοντος υλοποίησης Spark αλλά και με την χρήση αυτού ώστε να γίνει εκτέλεση των δύο αυτών υλοποιήσεων σε μια συστάδα υπολογιστών με διαφορετικό αριθμό από κόμβους και να γίνει σύγκριση των χρόνων εκτέλεσης ώστε να εξαχθούν συμπεράσματα σχετικά με το κατά πόσο αξίζει ένας αλγόριθμος εξόρυξης διαδιακασιών, όπως ο αλγόριθμος Alpha, να υλοποιηθεί με την χρήση του περιβάλλοντος υλοποίησης Spark.

1.3 Συνεισφορά της διπλωματικής εργασίας

Κατά την διάρκεια της εκπόνησης της παρούσας διπλωματικής εργασίας έγινε μελέτη της θεωρείας της εξόρυξης διαδικασιών. Συγκεκριμένα μελετήθηκε βιβλιογραφία αλλά και διάφορα μαθήματα μέσω διαδικτύου πάνω στο συγκεκριμένο αντικείμενο της επιστήμης των υπολογιστών. Ακόμα μελετήθηκε εκτενώς ο αλγόριθμος Alpha καθώς και η θεωρεία

για τα γραφήματα PetriNet. Σε επόμενο στάδιο έγινε η μελέτη του περιβάλλοντος υλοποίησης εφαρμογών Apache Spark αλλά και η μελέτη της γλώσσας προγραμματισμού Scala ώστε να γίνει η συγγραφή των δύο υλοποιήσεων του αλγόριθμου Alpha που εξετάστηκαν και μετρήθηκαν. Τα πειράματα έγιναν στην πλατφόρμα Databricks που παρέχει την δυνατότητα για δημιουργία μιας συστάδας υπολογιστών με διαφορετικό αριθμό από κόμβους. Εκεί εκτελέστηκαν μετρήσεις πάνω σε δεδομένα από έναν μεγάλο τηλεπικοινωνιακό οργανισμό της Ελλάδας. Από αυτές τις μετρήσεις έγινε φανερό ότι υπάρχει μεγάλη διαφορά στον χρόνο εκτέλεσης του αλγόριθμου Alpha σε Spark έκδοση σε σχέση με την υλοποίηση σε Scala. Στην Spark έκδοση ακόμα και για μικρό αριθμό από κόμβους η υλοποίηση είναι πιο γρήγορη σε σχέση με την ίδια υλοποίηση σε απλή Scala χωρίς την χρήση του Spark.

1.4 Δομή και οργάνωση της διπλωματικής εργασίας

Η δομή της παρούσας διπλωματικής εργασίας είναι ως εξής. Στο πρώτο κεφάλαιο παρουσιάζεται η βασική θεωρεία γύρω από την εξόρυξη δεδομένων αλλά και την εξόρυξη διαδικασιών. Στο δεύτερο κεφάλαιο παρουσιάζονται ο αλγόριθμος Alpha και τα δίκτυα Petri που είναι το παραγόμενο αποτέλεσμα της εκτέλεσης του αλγόριθμου Alpha. Στο τρίτο κεφάλαιο γίνεται μια εισαγωγή στην γλώσσα προγραμματισμού Scala, η οποία είναι η γλώσσα που χρησιμοποιήθηκε για την υλοποίηση του αλγόριθμου Alpha. Στο τέταρτο κεφάλαιο παρουσιάζεται το περιβάλλον ανάπτυξης εφαρμογών Spark το οποίο και χρησιμοποιήθηκε κατά την εκπόνηση της παρούσας διπλωματικής εργασίας ώστε να επιτευχθεί η παράλληλη εκτέλεση του αλγόριθμου Alpha σε διάφορους κόμβους μιας συστάδας υπολογιστών. Στο πέμπτο κεφάλαιο παρουσιάζονται αναλυτικά και επεξηγούνται αποσπάσματα του κώδικα υλοποίησης του αλγόριθμου Alpha αλλά και ο τρόπος με τον οποίο έγινε επιβεβαίωση της ορθής λειτουργίας του υλοποιημένου αλγόριθμου. Στο έκτο κεφάλαιο παρουσιάζονται το υπολογιστικό περιβάλλον στο οποίο εκτελέστηκαν πειράματα πάνω στον υλοποιημένο αλγόριθμο Alpha, τα δεδομένα της πειραματικής αξιολόγησης αλλά και τα πειραματικά αποτελέσματα. Στο έβδομο και τελευταίο κεφάλαιο παρουσιάζονται τα συμπεράσματα και οι προοπτικές της παρούσας διπλωματικής εργασίας.

2. Μοντέλα και Ανακάλυψη Διαδικασιών

Στο κεφάλαιο αυτό παρουσιάζεται μια εισαγωγή στην ανακάλυψη διαδικασιών και στα μοντέλα διαδιακασιών που είναι το παραγώμενο αποτέλεσμα κάθε αλγορίθμου εξόρυξης διαδικασιών. Ακόμα γίνεται και μια λεπτομερής παρουσίαση του αλγόριθμου Alpha ο οποίος υλοποιήθηκε στα πλαίσια της παρούσας διπλωματικής εργασίας.

2.1 Εισαγωγή στα Μοντέλα διαδικασιών

Σκοπός αυτού του κεφαλαίου είναι να γίνει η παρουσίαση του αλγορίθμου **Alpha** που υλοποιήθηκε στα πλαίσια αυτής της εργασίας. Ο συγκεκριμένος αλγόριθμος επιτρέπει σε μόλις 8 βήματα την μετατροπή δεδομένων γεγονότων (**event data**) σε μοντέλα διαδικασιών τα οποία έχουν την δυνατότητα να εκφράσουν μια μεγάλη γκάμα συμπεριφορών, όπως ακολουθίες (**sequences**), παραλληλισμό (**concurrency**), επιλογές (**choices**) και επαναλήψεις (**loops**). Φυσικά αυτός ο αλγόριθμος έχει κάποιους περιορισμούς στο τι μπορεί να κάνει, αλλά απεικονίζει τους βασικούς μηχανισμούς που χρησιμοποιούνται από πιο εξελιγμένους αλγορίθμους ανακάλυψης διαδικασιών (**process discovery**) , όπως ο αλγόριθμος **Inductive miner**, ο αλγόριθμος **Heuristic miner** και ο αλγόριθμος **Fuzzy miner**. Για να γίνει όμως κατανοητός ο τρόπος λειτουργίας του αλγόριθμου **Alpha** πρέπει πρώτα να γίνουν κατανοητά τα βασικά της μοντελοποίησης διαδικασιών (**process modeling**) και των δικτύων **Petri**.

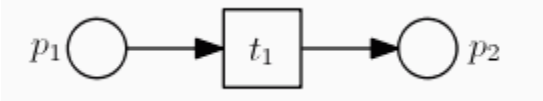
Υπάρχει μεγάλη ποικιλία από μοντέλα διαδικασιών τόσο στην βιομηχανία όσο και στην πανεπιστημιακή έρευνα. Υπάρχουν κάποια μοντέλα χαμηλότερου επιπέδου τα οποία δεν μπορούν να απεικονήσουν παραλληλισμό αλλά και κάποια συγκεκριμένα δομικά στοιχεία αλλά και κάποια άλλα υψηλότερου επιπέδου μοντέλα τα οποία έχουν καλύτερες δυνατότητες απεικόνησης διαδικασιών, όπως τα διαγράμματα BPMN, τα διαγράμματα UML, δίκτυα Petri κτλπ. Επιπλέον, τα διάφορα μοντέλα συνήθως βρίσκουν εφαρμογή σε διαφορετικά πεδία, παράδειγμα οι επιχειρηματικοί αναλυτές χρησιμοποιούν διαγράμματα BPMN και οι μηχανικοί λογισμικού χρησιμοποιούν διαγράμματα UML. Υπάρχει πληθώρα μοντέλων που χρησιμοποιούνται για να εκφράσουν το ίδιο ακριβώς αντικείμενο πράγμα το οποία δημιουργεί σύγχυση.

Αυτή η πληθώρα στα μοντέλα διαδικασιών οφείλεται στο ότι ο καθένας μπορεί να υλοποιήσει ένα σύνολο από νέα σύμβολα και να δημιουργήσει μια νέα γλώσσα για εμπορικούς κυρίως λόγους. Αλλά όμως αν κοιτάξουμε στα θεμέλια της δυναμικής συμπεριφοράς που θέλουμε να απεικονήσουμε, υπάρχουν κάποιες βασικές κοινές έννοιες. Αυτό μπορεί να γίνει κατανοητό από την ανακάλυψη μοντέλων διαδικασιών από γεγονότα (**event data**) όπως με τον αλγόριθμο **Alpha**. Τα γεγονότα είναι μερικώς ταξινομημένα (**ordered**) και αναφέρονται σε στιγμιότυπα διαδικασιών (**traces**) και δραστηριοτήτων. Βασιζόμενοι στα παραπάνω, κάποιος αλγόριθμος μπορεί να κατασκευάσει ένα μοντέλο διαδικασίας από ακολουθίες (**sequence**), επιλογές (**choice**), παραλληλισμούς (**parallel**) και επαναλήψεις (**iteration**) σε όποιο τύπο μοντέλου διαδικασίας επιθυμεί.

Πολλοί αλγόριθμοι εξόρυξης διαδικασιών χρησιμοποιούν δίκτυα Petri για την απεικόνηση του αποτελέσματος του αλγορίθμου αλλά και για την συμμόρφωση πάνω σε ένα μοντέλο. Αλλά αυτό δεν σημαίνει ότι οι από άκρη σε άκρη χρήστες θα χρεαστεί να δουν απαραίτητα ένα δίκτυο **Petri**. Εργαλεία όπως το ProM μπορούν να μετατρέψουν ένα δίκτυο **Petri** σε άλλους τύπους μοντέλων όπως τα διαγράμματα BPMN. Στο επόμενο κεφάλαιο παρουσιάζεται με λεπτομέρειες το μοντέλο απεικόνησης διαδικασιών **Petri**, το οποίο χρησιμοποιήθηκε στην συγκεκριμένη εργασία.

2.2 Δίκτυα Petri

Τα δίκτυα **Petri** αποτελούν μια τεχνική περιγραφής και ανάλυσης κατανεμημένων (**distributed**) συστημάτων και διαδικασιών. Έχουν ένα εκφραστικό γραφικό περιβάλλον και αποτελούν επέκταση της μαθηματικής θεωρείας αυτομάτων (**Automata theory**). Ένα δίκτυο **Petri** είναι ένα κατευθυνόμενο διμερές γράφημα (**directed bipartite graph**), το οποίο αποτελείται από κύκλους/καταστάσεις (**places/states**) και τετράγωνα/μεταβάσεις (**squares/transitions**). Τα τετράγωνα απεικονίζουν τις μεταβάσεις από κάποια δραστηριότητα (**activity**) σε κάποια άλλη.



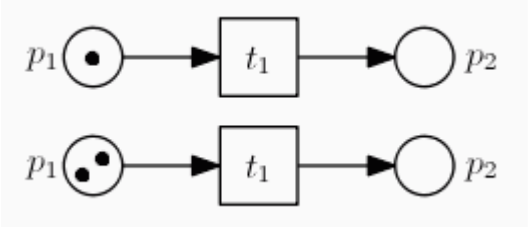
Εικόνα 7:
Αναπαράσταση ενός δικτύου Petri

Οι **καταστάσεις** μπορούν να είναι είσοδοι ή έξοδοι σε κάποιες μεταβάσεις του συστήματος. Οι **καταστάσεις** απεικονίζουν την κατάσταση (**state**) του συστήματος. Οι μεταβάσεις απεικονίζουν τις αλλαγές της κατάστασης του συστήματος.

Ένα δίκτυο **Petri** αποτελείται από κύκλους, τετράγωνα και πλευρές (arcs) και σαν δίκτυο **Petri** μπορεί να οριστεί μια τριάδα **(P, T, F)** όπου

- **P** είναι ένα πεπερασμένο σύνολο από καταστάσεις.
- **T** είναι ένα πεπερασμένο σύνολο από μεταβάσεις**.** Το σύνολο αυτό είναι και το σύνολο των γεγονότων σε ένα μοντέλο.
- **F** είναι το σύνολο των ακμών σε ένα δίκτυο **Petri**, δηλαδή ένα υποσύνολο ακμών. Παράδειγμα για την εικόνα 7 το πεπερασμένο σύνολο ακμών F είναι το F= { (p_1 , t_1) , (t_1 , p_2)}

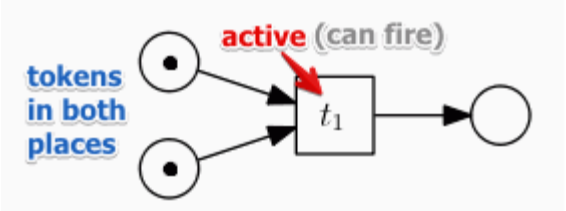
Κάθε κατάσταση μπορεί να περιέχει ένα ή περισσότερα ενδείξεων (**tokens**). Μια ένδειξη είναι μια μονάδα εργασίας που πρέπει να εκτελεστεί.



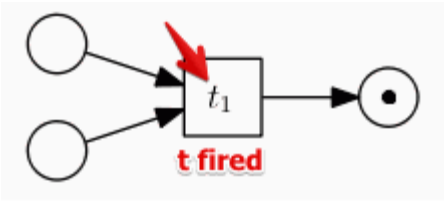
Εικόνα 8:
Αναπαράσταση των ενδείξεων. Στο πρώτο σχήμα υπάρχει μια ένδειξη ενώ στο δεύτερο δύο.

Στιγμιότυπο **(marking)** είναι η κατάσταση ενός δικτύου Petri που αποτυπώνει την κατανομή των ενδείξεων σε όλες τις καταστάσεις του δικτύου Petri. Μια μετάβαση (**transition**) αλλάζει την κατάσταση (**state**) ενός Petri Net με την πυροδότηση (**firing**).

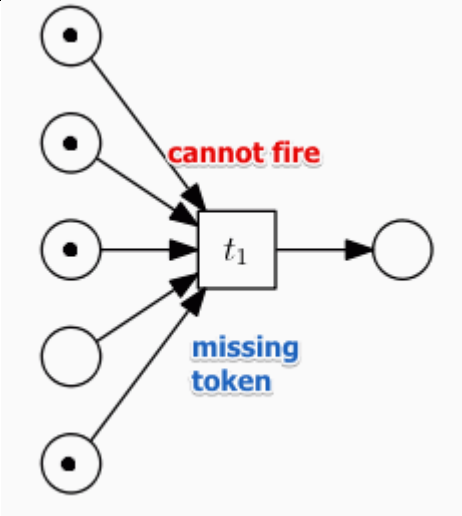
Ας δούμε τα παρακάτω παραδείγματα.



Εικόνα 9:
Στην παραπάνω εικόνα φαίνεται ότι η πυροδότηση είναι εφικτή, καθώς υπάρχουν δύο ενδείξεις, μια σε κάθε είσοδο της μετάβασης t_1. Το αποτέλεσμα φαίνεται στην παρακάτω εικόνα όπου οι είσοδοι της μετάβασης t_1 δεν έχουν πλέον ενδείξεις, και η έξοδος της μετάβασης t_1 έχει ακριβώς μια ένδειξη.



Εικόνα 10:
Δηλαδή, σύμφωνα με τα παραπάνω για να είναι δυνατή η πυροδότηση μιας μετάβασης σε ένα δίκτυο Petri είναι απαραίτητη η ύπαρξη μιας τουλάχιστον ένδειξης σε κάθε είσοδο της μετάβασης. Ακόμα και στις εξόδους του δικτύου Petri είναι δυνατό να δημιουργηθούν τόσες ενδείξης όσες είναι οι έξοδοι της μετάβασης.



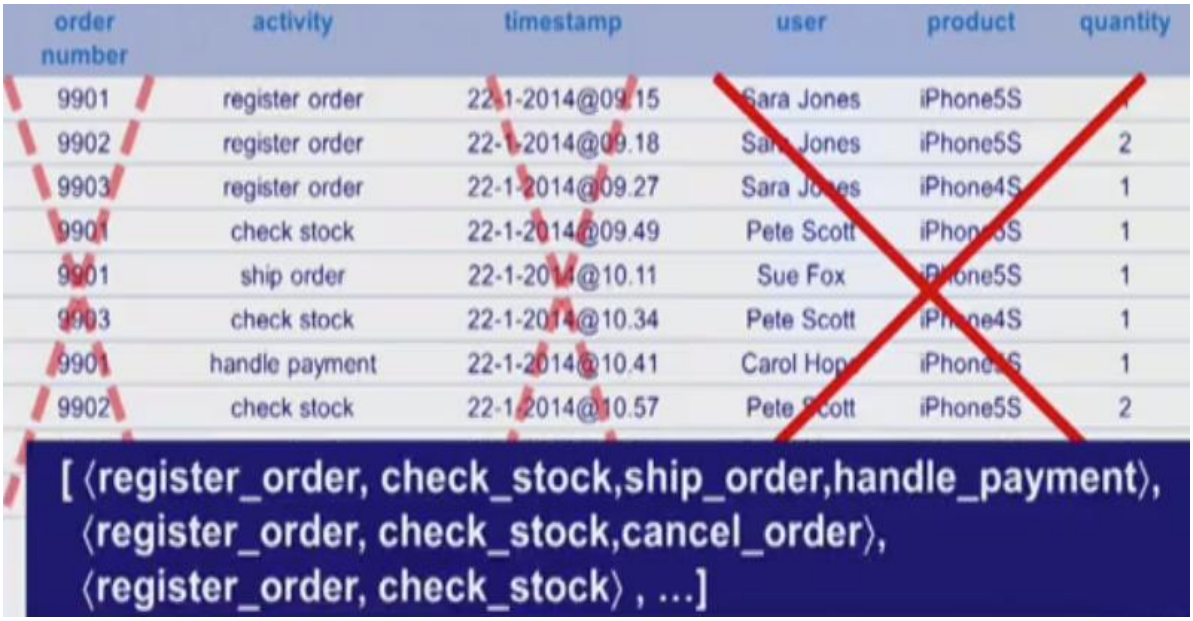
Εικόνα 11:
Στην παρακάνω εικόνα φαίνεται μια κατάσταση σε ένα δίκτυο Petri όπου δεν μπορεί να γίνει η πυροδότηση μιας μετάβασης καθώς λείπει μια ένδειξη η οποία έπρεπε να είναι διαθέσιμη.

2.3 Αλγόριθμος Alpha– Δομικά στοιχεία

Ο αλγόριθμος **Alpha** είναι ένας αλγόριθμος ανακάλυψης διαδικασιών (**process discovery**) από δεδομένα γεγονότων. Σύμφωνα με τα όσα είδαμε στο κεφάλαιο 1.1, είναι κατανοητό ότι η διαδικασία της εύρεσης διαδικασίας ανήκει στην κατηγορία **Play-In**. Έτσι από ένα σύνολο από γεγονότα (**event log**), ο αλγόριθμος **Alpha** θα ανακαλύψει αυτόματα ένα μοντέλο διαδικασίας και συγκεκριμένα ένα δίκτυο **Petri**. Ο αλγόριθμος **Alpha** ήταν ο πρώτος αλγόριθμος ο οποίος ήταν ικανός να ανακαλύψει παράλληλα γεγονότα (**concurrency**), να ανακαλύψει επαναλήψεις (**loops**) αλλά και επιλογές στην διαδικασία (**choices**) ενώ ταυτόχρονα μπορεί και εγγυάται κάποιες συγκεκριμένες ιδιότητες.

Αλλά τι είδους είσοδο χρησιμοποιεί ο αλγόριθμος **Alpha**;

Συγκεκριμένα σε σύνολο δεδομένων (**data set**), ο αλγόριθμος **Alpha** αγνοεί τις πηγές και άλλα δεδομένα. Ακόμα αγνοεί και τις χρονικές στιγμές που συνέβησαν τα γεγονότα, όπως και τον κωδικό (**caseID)** των συνόλων από δραστηριότητες που επεξεργάζεται. Το μόνο που ενδιαφέρει τον αλγόριθμο Alpha για να παράγει αποτελέσματα είναι η ταξινόμηση (**ordering**) των γεγονότων. Έτσι ένα πρώτο βήμα για τον αλγόριθμο **Alpha** είναι η μετατροπή του συνόλου γεγονότων σε ένα ίχνος δραστηριοτήτων (**traces**), όπου ένα **trace** είναι μια ακολουθία από ονόματα δραστηριοτήτων (**activity names**).



order number	activity	timestamp	user	product	quantity
9901	register order	22-1-2014@09.15	Sara Jones	iPhone5S	1
9902	register order	22-1-2014@09.18	Sara Jones	iPhone5S	2
9903	register order	22-1-2014@09.27	Sara Jones	iPhone4S	1
9901	check stock	22-1-2014@09.49	Pete Scott	iPhone5S	1
9901	ship order	22-1-2014@10.11	Sue Fox	iPhone5S	1
9903	check stock	22-1-2014@10.34	Pete Scott	iPhone4S	1
9901	handle payment	22-1-2014@10.41	Carol Hop	iPhone5S	1
9902	check stock	22-1-2014@10.57	Pete Scott	iPhone5S	2

[⟨register_order, check_stock,ship_order,handle_payment⟩,
⟨register_order, check_stock,cancel_order⟩,
⟨register_order, check_stock⟩ , …]

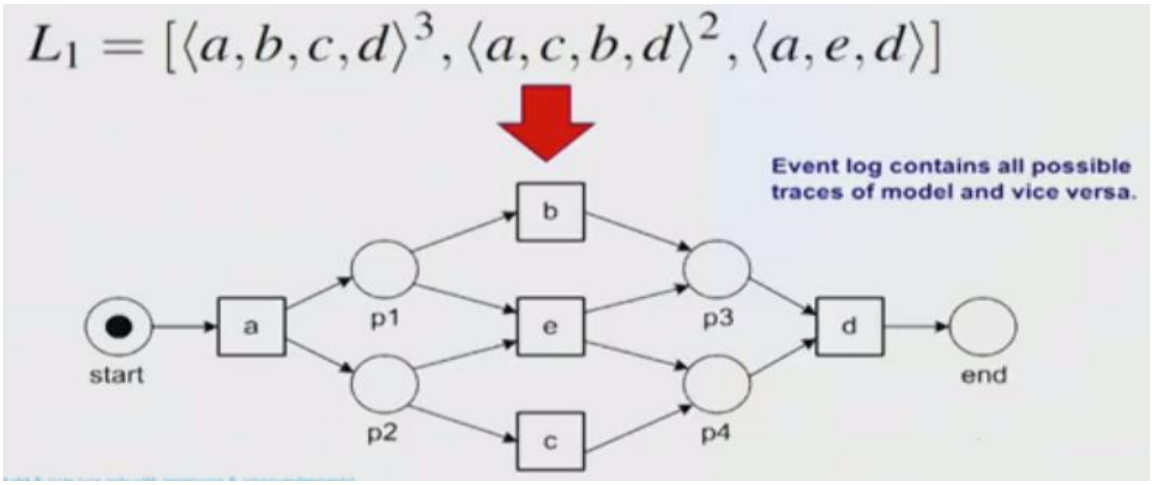
Εικόνα 12:
Στην εικόνα αυτή φαίνεται ένα σύνολο από γεγονότα που βασίζεται σε δεδομένα από την πραγματική ζωή. Ακόμα γίνεται κατανοητό ότι ο αλγόριθμος Alpha εδιαφέρεται μόνο για τις δραστηριότητες και όχι για άλλες πληροφορίες. Ακόμα βλέπουμε πως γίνεται η κατασκευή των ιχνών δραστηριοτήτων από το order number της εικόνας.

Στην συνέχεια φαίνεται παρακάτω ένα πιο απλό παράδειγμα.

L_1 = [$(a,b,c,d)^3$, $(a,c,b,d)^2$, $(a,e,d)^1$]

Εδώ βλέπουμε το σύνολο δεδομένων L_1, το οποίο αποτελείται από 6 ίχνη δραστηριοτήτων, τα οποία μπορούμε να τις θεωρήσουμε σαν 6 διαφορετικές περιπτώσεις (**cases**). Αυτές οι περιπτώσεις μοντελοποιούνται σαν μια ακολουθία από δραστηριότητες. Έτσι η ακολουθία a,b,c,d εκτελέστηκε 3 φορές και έτσι υπάρχουν 3 ίχνη δραστηριοτήτων αυτού του τύπου. Έτσι ένα σύνολο γεγονότων είναι ένα πολύ-σύνολο (**multiset**) από ίχνη δραστηριοτήτων επειδή το ίδιο ίχνος δραστηριοτήτων μπορεί να εμφανιστεί πολλαπλές φορές, και ένα ίχνος δραστηριοτήτων είναι μια ακολουθία από δραστηριότητες οι οποίες μπορούν να απομονωθούν σε σχέση με όλες τις υπόλοιπες ιδιότητες.

Σκοπός του αλγορίθμου **Alpha** είναι αν τροφοδοτήσουμε κάποιο σύνολο γεγονότων όπως το L_1, πρέπει αυτόματα να μάθουμε ένα μοντέλο διαδικασίας (**process model**) το οποίο θα απεικονίζει ακριβώς την συμπεριφορά του σύνολο γεγονότων L_1. Στην παρακάτω εικόνα φαίνεται το δίκτυο Petri που εξάγει ο αλγόριθμος **Alpha** για το σύνολο γεγονότων L_1. Φυσικά το αποτέλεσμα θα μπορούσε να ήταν και οποιαδήποτε άλλη μορφή αναπαράστασης μοντέλων , όπως ένα διάγραμμα BPMN ή ένα διάγραμμα UML, καθώς η κύρια ευθύνη του αλγορίθμου είναι να καταλάβει την συμπεριφορά μέσα από το σύνολο γεγονότων**.**



Εικόνα 13:
Το δίκτυο Petri που παράγει ο αλγόριθμος Alpha για το σύνολο γεγονότων L_1

Το σημείο εκκίνησης του αλγόριθμου **Alpha** είναι οι σχέσεις ταξινόμησης (**ordering relations**). Ο αλγόριθμος **Alpha** δεν έχει κανένα ενδιαφέρον πάνω στις συχνότητες που εμφανίζονται τα ίχνη δραστηριοτήτων μέσα στο σύνολο γεγονότων, όπως και για οποιαδήποτε άλλα χαρακτηριστικά που έχουν αποθηκευτεί για κάθε γεγονός. Αυτό λοιπόν που ενδιαφέρει τον αλγόριθμο **Alpha** είναι να βρει μέσω των γεγονότων ποιες δραστηριότητες ακολουθούν ποιες. Για παράδειγμα για την ακολουθία a,b,c,d, ο αλγόριθμος θα δει ότι το a ακολουθείται από το b, το b από το c και το c από το d. Αυτή η σχέση ονομάζεται απευθείας διαδοχή (**direct succession**) και ισχύει όταν κάποια δραστηριότητα x ακολουθείται απευθείας από το y και συμβολίζεται σαν **x>y**.

Για παράδειγμα στο σύνολο γεγονότων L_1 βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία απευθείας διαδοχής**.**

(a>b , a>c , a>e , b>c , b>d , c>b , c>d , e>d)

Η δεύτερη σχέση που εξετάζει ο αλγόριθμος **Alpha** είναι η αιτιότητα (**causality**) και συμβολίζεται με ένα βέλος (**x->y**) και σημαίνει ότι το x ακολουθείται από το y, αλλά το y δεν ακολουθείται **ποτέ** από το x. Στο σύνολο γεγονότων L_1 βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία «αιτιότητα»**.**

(a->b , a->c , a->e , b->d , c->d , e->d)

Η τρίτη σχέση ισχύει όταν η σχέση απευθείας διαδοχής είναι έγκυρη και για τις δύο κατευθύνσεις, τότε λέμε ότι η σχέση είναι παράλληλη (**parallel**). Άρα όταν μερικές φορές το x ακολουθείται από το y και μερικές φορές το y ακολουθείται από το x τότε το x είναι παράλληλο με το y (**x||y**). Στο σύνολο γεγονότων L_1 βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **parallel.**

(b||c , c||b)

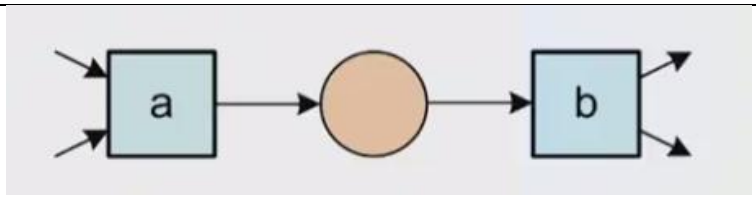
Η τέταρτη και τελευταία σχέση ισχύει όταν το x δεν ακολουθείται ποτέ απευθείας από το y και το y δεν ακολουθείται ποτέ απευθείας από το x (**x#y**). Τότε λέμε ότι αυτό είναι μια σχέση επιλογής (**choice relation)**. Στο σύνολο γεγονότων L_1 βρίσκουμε τις παρακάτω σχέσεις που ανήκουν στην κατηγορία **«**επιλογής**».**

(b#e , e#b , c#e , a#d …)

Άρα συνοψίζοντας έχουμε τις παρακάτω σχέσεις

- **Direct succession (x>y)** Υπάρχουν υπό-ακολουθίες …xy….
- **Causality (x->y)** Υπάρχουν υπό-ακολουθίες …xy…. Αλλά όχι …yx….
- **Parallel (x||y)** Υπάρχουν υπό-ακολουθίες …xy…. και …yx….
- **Choice- exclusiveness** (**x#y**) Δεν υπάρχουν υπό-ακολουθίες …xy…. ούτε και …yx….

Αυτές οι παραπάνω σχέσεις χρησιμοποιούνται για να εξαγωγή διάφορων μοτίβων (**patterns**) στην διαδιακασία. Στην παρακάτω εικόνα (**Εικόνα 14**) φαίνεται η σχέση **«αιτιότητας»**. Δηλαδή αυτό το μοτίβο περιμένουμε να δούμε σε ένα δίκτυο Petri όταν μερικές φορές το a ακολουθείται από το b, αλλά ποτέ δεν συμβαίνει το αντίθετο.

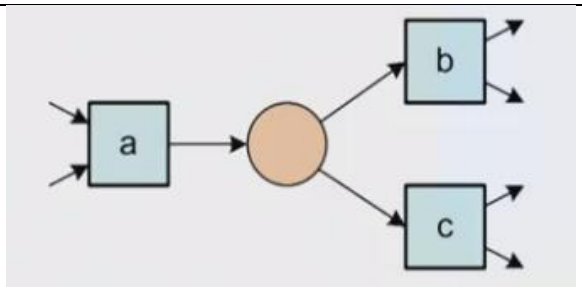


Εικόνα 14:
Causality pattern (a->b)

Στην παρακάτω (**Εικόνα 15**) φαίνεται το μοτίβο **XOR-split** σε ένα δίκτυο Petri που προκύπτει

- όταν το a μερικές φορές ακολουθείται από το b, αλλά ποτέ όμως αντίστροφα **(a->b)**
- όταν το a μερικές φορές ακολουθείται από το c, αλλά ποτέ όμως αντίστροφα **(a->c)**
- όταν το b και το c δεν ακολουθεί ποτέ το ένα το άλλο (**b#c**)

Αυτό το μοτίβο σημαίνει ότι μετά το a, υπάρχει μια επιλογή (**choice**) μεταξύ των b και c.

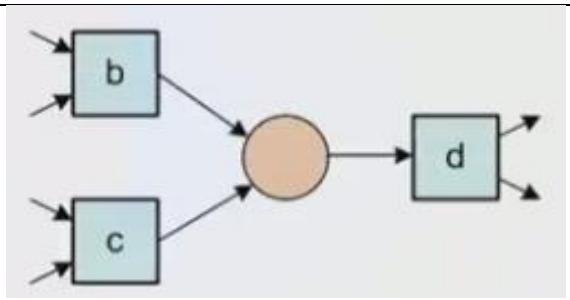


Εικόνα 15:

XOR-split pattern (a->b , a->c , b#c) (επιλογή ανάμεσα στο **b** και **c)**

Το αντίστοιχο μοτίβο του **XOR-split** είναι το **XOR-join,** όπως φαίνεται στην εικόνα 16. Ισχύει όταν

- το b έχει σχέση **«αιτιότητας»** με το d (**b->d**)
- το c έχει σχέση **«αιτιότητας»** με το d (**c->d**)
- το b με το c ποτέ δεν ακολουθεί το ένα το άλλο (**b#c**)

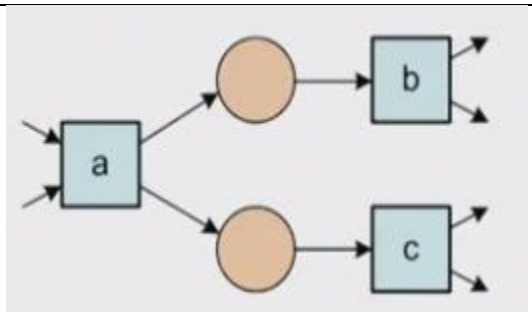


Εικόνα 16:
XOR-join pattern (b->d , c->d , b#c)

Ως προς την παραλληλία (**concurrency**) σε ένα δίκτυο Petri υπάρχουν δύο μοτίβα, το **AND-split** και το **AND-join**.

Στην παρακάτω (**Εικόνα 17**) φαίνεται το μοτίβο **AND -split** σε ένα δίκτυο Petri που προκύπτει

- όταν το a μερικές φορές ακολουθείται από το b, αλλά ποτέ όμως αντίστροφα **(a->b)**
- όταν το a μερικές φορές ακολουθείται από το c, αλλά ποτέ όμως αντίστροφα **(a->c)**
- όταν το b και το c δεν ακολουθεί ποτέ το ένα το άλλο (**b||c**)

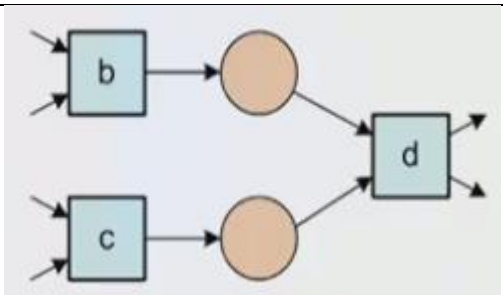


Εικόνα 17:
AND-split pattern (a->b ,a->c , b||c)

Το αντίστοιχο pattern του **AND-split** είναι το **AND-join,** όπως φαίνεται στην **εικόνα 18**. Ισχύει όταν

- το b έχει σχέση **«αιτιότητας»** με το d (**b->d**)

- το c έχει σχέση **«αιτιότητας»** με το d (**c->d**)
- το b με το c ποτέ δεν ακολουθεί το ένα το άλλο (**b||c**)



Εικόνα 18:
AND-join pattern (b->d ,c->d , b||c)

Άρα βασιζόμενοι στα παραπάνω μοτίβα είναι δυνατή η κατασκευή ενός δικτύου Petri. Έτσι λοιπόν όλες οι σχέσεις ταξινόμησης (**ordering relations**) που εξάγονται από ένα σύνολο γεγονότων, όπως το L_1, αποτελούν ένα πίνακα σχέσεων (ονομαζόμενο ως footprint) και αυτός είναι ακριβώς ο πίνακας στην Εικόνα 19 για σύνολο γεγονότων L_1. Κάθε κελί (**cell**) σε αυτόν τον πίνακα έχει μια από τις τέσσερεις σχέσεις που παρασουσιάστηκαν νωρίτερα (**απευθείας διαδοχής, αιτιότητας, παραλληλίας, επιλογής**). Έτσι λοιπόν σύμφωνα με τον πίνακα είναι δυνατόν να έχουμε σχέση **αιτιότητας** ως προς την μία κατεύθυνση αλλά και ως προς την άλλη.

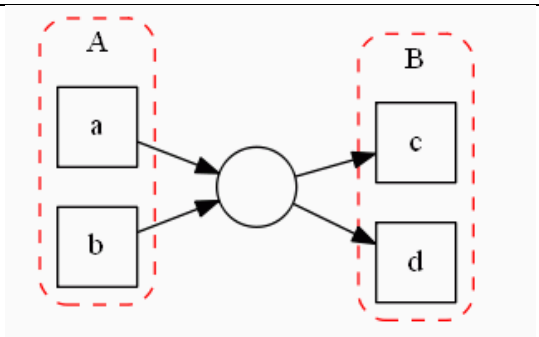
	a	b	c	d	e
a	$\#_{L_1}$	$\rightarrow_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$
b	$\leftarrow_{L_1}$	$\#_{L_1}$	$\|\|_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
c	$\leftarrow_{L_1}$	$\|\|_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$
d	$\#_{L_1}$	$\leftarrow_{L_1}$	$\leftarrow_{L_1}$	$\#_{L_1}$	$\leftarrow_{L_1}$
e	$\leftarrow_{L_1}$	$\#_{L_1}$	$\#_{L_1}$	$\rightarrow_{L_1}$	$\#_{L_1}$

Εικόνα 19:
Footprint graph

2.4 Αλγόριθμος Alpha– Βήματα

Ο αλγόριθμος **Alpha** είναι ένας πολύ βασικός αλγόριθμος που περίεχει μόνο 8 βήματα αλλά μοιάζει πολύπλοκος όταν κάποιος τον μελετήσει για πρώτη φορά. Παρά την απλότητα του αλγόριθμου **Alpha** μπορεί να ανακαλύψει ένα τεράστιο εύρος από διαφορετικές διαδικασίες. Διαδικασίες οι οποίες περιέχουν παραλληλισμό (concurrency), διαδικασίες οι οποίες περιέχουν επαναλήψεις (**loops**) και διαδικασίες οι οποίες περιέχουν επιλογές (**choices**). Έτσι λοιπόν, η εκτέλεση του αλγόριθμου **Alpha** ξεκινάει από ένα σύνολο από ίχνη δραστηριοτήτων με σκοπό την κατασκευή ενός Petri Net.

- **Πρώτο Βήμα**
  Στο πρώτο βήμα ο αλγόριθμος **Alpha** σκανάρει το σύνολο γεγονότων για να δει ποιες δραστηριότητες ή μεταβάσεις λαμβάνουν μέρος. Άρα ψάχνει για το ποιες μοναδικές δραστηριότητες υπάρχουν στο σύνολο γεγονότων.
- **Δεύτερο Βήμα**
  Στο δεύτερο βήμα ο αλγόριθμος **Alpha** βρίσκει ένα σύνολο από όλες τις αρχικές δραστηριότητες που υπάρχουν στο σύνολο γεγονότων**,** δηλαδή το πρώτο στοιχείο κάθε ίχνους δραστηριότητας που ανήκει στο σύνολο γεγονότων.
- **Τρίτο Βήμα**
  Στο τρίτο βήμα ο αλγόριθμος **Alpha** βρίσκει ένα σύνολο από όλες τις τελικές δραστηριότητες που υπάρχουν στο σύνολο γεγονότων**,** δηλαδή το τελευταίο στοιχείο κάθε ίχνους δραστηριοτήτων που ανήκει στο σύνολο γεγονότων.
- **Τέταρτο Βήμα**
  Στο τέταρτο βήμα ο αλγόριθμος **Alpha** κατασκευάζει τον πίνακα σχέσεων**,** δηλαδή τον πίνακα που περιέχει όλες τις σχέσεις μεταξύ των διάφορων δραστηριοτήτων του συνόλου γεγονότων. Η διαδικασία της ανακάλυψης μιας διαδιακασίας έχει να κάνει κυρίως με την ανακάλυψη μοτίβων / καταστάσεων πάνω σε ένα παραγώμενο δίκτυο Petri στην περίπτωση του αλγόριθμου **Alpha.** Για την εύρεση των καταστάσεων πρέπει να γίνει ο προσδιορισμός ενός συνόλου από μεταβάσεις (**transitions**) A και B, όπου το A είναι η μετάβαση εισόδου για μια κατάσταση και B είναι η μετάβαση εξόδου για την κατάσταση.
  Μετά την κατασκευή του πίνακα σχέσεων πρέπει σύμφωνα με τις πληροφορίες που παρέχει το πίνακας σχέσεων να βρεθούν δύο σύνολα από δραστηριότητες X και Y, τα οποία ονομάζονται **causal groups**. Οι δραστηριότητες αυτές πρέπει να έχουν την ακόλουθη ιδιότητα. Αν επιλεγούν δύο δραστηριότητες από το σύνολο X δεν πρέπει ποτέ να ακολουθεί η μία την άλλη, δηλαδή πρέπει να ισχύει η σχέση “επιλογής”. Το ίδιο ακριβώς πρέπει να ισχύει και για το σύνολο Y. Το δεύτερο προαπαιτούμενο είναι ότι αν επιλεγεί μια δραστηριότητα από το X και μια δραστηριότητα από το Y πρέπει να έχουν μια σχέση «αιτιότητας» μεταξύ τους. Αυτό φυσικά θα πρέπει να ισχύει για όλους τους συδιασμούς από ζεύγη για όλες τις δραστηριότητες από το X στο Y. Δηλαδή πρέπει να βρεθούν ζεύγη (X,Y) από σύνολα από δραστηριότητες τέτοια ώστε κάθε στοιχείο x∈X και κάθε στοιχείο y∈Y να ανήκουν στη σχέση «αιτιότητας» (x->y) και τα στοιχεία των X και Y να είναι ανεξάρτητα μεταξύ τους (x_1 # x_2) και (y_1 # y_2). Αυτό φαίνεται ξεκάθαρα και στην **Εικόνα 20**.



Εικόνα 20:

Εύρεση όλων των ζεύγων από σύνολα (A,B) για τα οποία ισχύει

- Κάθε $t_A \in$ A πρέπει να συνδέεται με όλα τα $t_B \in$ B για κάθε κατάσταση p
- ∀a ∈ A και $\forall b \in$ B πρέπει να ισχύει a->b
- $\forall a_1, a_2 \in$ A : $a_1 \# a_2$ και $\forall b_1, b_2 \in$ B : $b_1 \# b_2$

Στο συγκεκριμένο παράδειγμα τα 2 παράπανω σύνολα προέκυψαν από τις σχέσεις a#b , c#d , a->c , a->d , b->c , b->d και όλες οι μεταβάσεις από το A τοποθετούν ενδείξεις στην κατάσταση p και οι μεταβάσεις από το B καταναλώνουν τις ενδείξεις από την κατάσταση p.

- **Πέμπτο Βήμα**

  Στο πέμπτο βήμα γίνεται διαγραφή από το σύνολο X_L των καταστάσεων που βρέθηκαν στο βήμα 4 όλων των ζευγών (**pairs**) τα οποία δεν είναι μέγιστα (**maximal)**. Δηλαδή από το αρχικό σύνολο X_L θα διατηρηθούν μόνο εκείνα που περιέχουν τον μέγιστο αιρθμό από στοιχεία που μπορούν να συνδεθούν μέσω μιας συγκεκριμένης κατάστασης (**Εικόνα 21**).

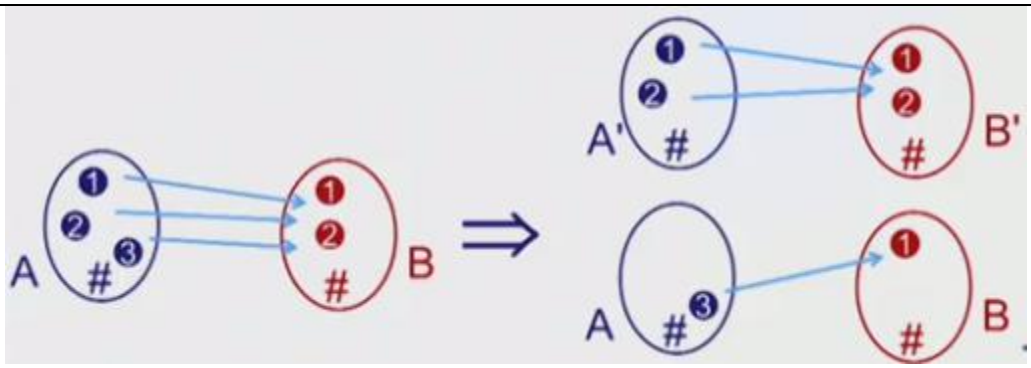
- **Έκτο Βήμα**

  Στο έκτο βήμα σχηματίζεται ένα σύνολο από καταστάσεις P_L. Από τα μέγιστα ζεύγη που βρέθηκαν στο βήμα 5 τα οποία είναι καταστάσεις του τελικού δικτύου Petri προσθέτουμε μια αρχική κατάσταση I και μια τελική κατάσταση O. Αυτό το σύνολο αποτελεί και το σύνολο P της τριάδας **(P, T, F)** που περιγράφει το δίκτυο Petri που θα παραχθεί.

- **Έβδομο Βήμα**

  Στο έβδομο βήμα καθορίζεται η ροή των ακμών (**flow relation)** του δικτύου Petri που θα παραχθεί, δηλαδή παράγονται τα βέλη (arcs) που υπάρχουν στο δίκτυο Petri. Από τα προηγούμενα βήματα του αλγορίθμου έχουν βρεθεί οι μεταβάσεις και οι καταστάσεις (στο πέμπτο βήμα). Έτσι στο έβδομο βήμα για κάθε κατάσταση p(A,B) ενώνεται για κάθε στοιχείο a που ανήκει στο σύνολο του A με την αντίστοιχη μετάβαση και κάθε στοιχείο b που ανήκει στο σύνολο του B με την αντίστοιχη μετάβαση. Επιπλέον σχεδιάζεται ένα τόξο από την αρχική κατάσταση (place) ι_L προς κάθε αρχίκη μετάβαση (start transition) $t \in T_i$ και ένα τόξο από κάθε τελική μετάβαση $t \in T_O$ προς την τελική κατάσταση o_L. Άρα σε αυτό το βήμα γίνονται οι συνδέσεις του δικτύου Petri αλλά όπως φαίνεται ξεκάθαρα η όλη πολυπλοκότητα του αλγορίθμου βρίσκεται στα βήματα 4 και 5. Το σύνολο που θα παραχθεί από το

βήμα 7 αποτελεί και το σύνολο **F** της τριάδας **(P, T, F)** που περιγράφει το δίκτυο Petri που θα παραχθεί.



Εικόνα 21:
Απεικόνηση του βήματος 5. Εύρεση των μέγιστων ζευγών. Αν το παραγώμενο από το βήμα 4 σύνολο X_L, περιέχει τα 3 παραπάνω ζεύγη συνόλων από δραστηριότητες τότε το βήμα 5 θα κρατήσει μόνο το συνόλο 1 (αριστερά) γιατί περιέχει και τα άλλα 2 σύνολα (δεξιά)

- **Όγδωο Βήμα**
  Στο όγδωο και τελευταίο βήμα του αλγόριθμου Alpha γίνεται η κατασκευή του τελικού δικτύου Petri το οποίο αποτελείται από μια τριάδα συνόλων **(P, T, F).** Το σύνολο **P** αποτελεί τις καταστάσεις του δικτύου Petri οι οποίες έχουν παραχθεί από το βήμα 5 του αλγορίθμου Alpha. Το σύνολο **T** αποτελεί τις μεταβιβάσεις του δικτύου Petri και είναι το σύνολο των μοναδικών δραστηριοτήτων που υπάρχουν στο εξεταζόμενο σύνολο γεγονότων. Το σύνολο **F** αποτελεί τα τόξα του δικτύου Petri τα οποία έχουν παραχθεί από το βήμα 7 του αλγόριθμου **Alpha**.

2.5 Αλγόριθμος Alpha – Παράδειγμα

Σε αυτό το κεφάλαιο θα παρουσιαστεί η εκτέλεση του αλγόριθμου **Alpha** για το σύνολο γεγονότων L_1 εκτελώντας όλα τα βήματα του αλγορίθμου ένα προς ένα

L_1 = [$(a,b,c,d)^3$, $(a,c,b,d)^2$, $(a,e,d)^1$]

- **Πρώτο Βήμα**
  Εύρεση όλων των μοναδικών δραστηριοτήτων ή μεταβάσεων**: T = {a,b,c,d,e}**
- **Δεύτερο Βήμα**
  Αρχικές δραστηριότητες : T_i = {a}
- **Τρίτο Βήμα**
  Τελικές δραστηριότητες: T_o = {e}

- **Τέταρτο Βήμα**

  Εύρεση πίνακα σχέσεων και πιθανών καταστάσεων του δικτύου Petri (X_L).

	a	**b**	**c**	**d**	**e**
a	#	->	->	#	->
b	<-	#	||	->	#
c	<-	||	#	->	#
d	#	<-	<-	#	<-
e	<-	#	#	->	#

X_L = {
({a} , {b}),
({a} , {c}),
({a} , {e}),
({a} , {b, e}),
({a} , {c, e}),
({b} , {d}),
({c} , {d}),
({e} , {d}),
({b,e} , {d}),
({c,e} , {d})
}

- **Πέμπτο Βήμα**

  Εύρεση των μέγιστων ζευγών από το σύνολο X_L που βρέθηκε στο βήμα 4. Παρατηρούμε ότι στο σύνολο X_L περιέχονται πολλά στοιχεία τα οποία δεν είναι μέγιστα και ορισμένα από αυτά περιέχονται σε άλλα στοιχεία. Παράδειγμα τα ({a} , {b}), ({a} , {c}), ({a} , {e}), ({b} , {d}), ({c} , {d}), ({e} , {d}) τα οποία θα διαγραφούν από το πέμπτο βήμα του αλγόριθμου **Alpha.** Έτσι το σύνολο των μέγιστων ζευγών είναι το Y_L = { ({a} , {b, e}), ({a} , {c, e}), ({b,e} , {d}), ({c,e} , {d}) }

- **Έκτο Βήμα**

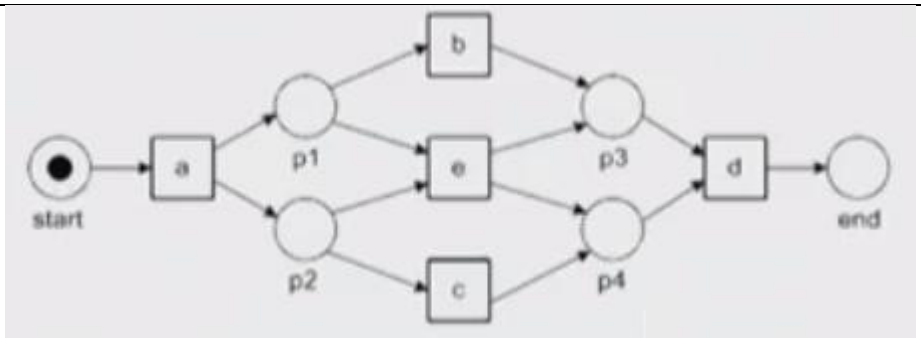
  Στο βήμα αυτό προστίθεται στο σύνολο Y_L μια αρχική και μια τελική κατάσταση και έτσι σχηματίζεται το σύνολο P της τριάδας **(P, T, F)** που περιγράφει το δίκτυο Petri που θα παραχθεί.

- **Έβδομο Βήμα**

  Σε αυτό το βήμα σχηματίζεται το σύνολο **F** της τριάδας **(P, T, F)** που περιγράφει το δίκτυο Petri που θα παραχθεί. Το σύνολο αυτό περιέχει όλα τα τόξα του δικτύου Petri.

- **Όγδωο Βήμα**

  Σε αυτό το βήμα σχηματίζεται το τελικό δίκτυο Petri που φαίνεται στην εικόνα 22.



Εικόνα 22:
Παραγόμενο δίκτυο Petri για το σύνολο γεγονότων L_1 = [$(a,b,c,d)^3$, $(a,c,b,d)^2$, $(a,e,d)^1$]

Στην **Εικόνα 22** βλέπουμε τις καταστάσεις που παρήχθησαν στο πέμπτο βήμα του αλγόριθμου **Alpha.** Συγκεκριμένα

- το p1 αντιστοιχεί στο ({a} , {b, e})
- το p2 αντιστοιχεί στο ({a} , {c, e})
- το p3 αντιστοιχεί στο ({b,e} , {d})
- το p4 αντιστοιχεί στο ({c,e} , {d})

Ακόμα βλέπουμε και την αρχική και τελική κατάσταση που προστέθηκαν στο έκτο βήμα του αλγόριθμου **Alpha.**

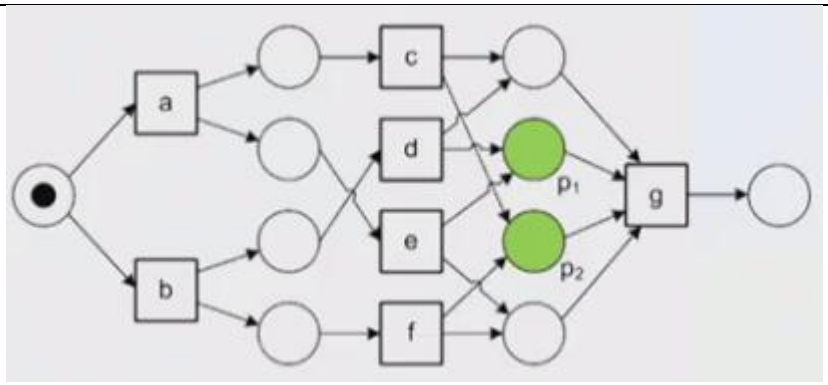
Συνοψίζοντας όλα τα παραπάνω, ο αλγόριθμος **Alpha** παρέχει μια βασική προσέγγιση στην ανακάλυψη διαδικασιών. Έχει όμως και πολλούς περιορισμούς οι οποίοι θα αναλυθούν στο επόμενο κεφάλαιο. Αλλά παρά τους περιορισμούς, ο αλγόριθμος αυτός είναι πολύ χρήσιμος και παρουσιάζει τα βασικά συστατικά κάθε άλλου αλγόριθμου ανακάλυψης διαδικασιών (**process discovery**), όπως την ανακάλυψη επαναλήψεων, παραλληλισμών, επιλογών κτλπ.

2.6 Αλγόριθμος Alpha – Περιορισμοί

Στο προηγούμενο κεφάλαιο είδαμε ότι ο αλγόριθμος **Alpha** αποτελείται από μόλις οκτώ βήματα και παρά την απλότητα του είναι αρκετά δυνατός ώστε να εξάγει διαδικασίες από μεγάλα σύνολα γεγονότων. Αλλά όμως δεν είναι αρκετά δυνατός ώστε να μπορέσει να ανταπεξέλθει σε όλες τις περιπτώσεις που η εφαρμογή του είναι χρήσιμη καθώς έχει κάποιους περιορισμούς και δεν μπορεί να παράγει τα αποτελέσματα που περιμένει ο χρήστης.

Ένα μειονέκτημα λοιπόν του αλγόριθμου **Alpha** είναι ότι δεν μπορεί να προσδιορίσει τα «υπονοούμενες καταστάσεις» (**implicit places)**. Οι «υπονοούμενες καταστάσεις» δεν είναι

απαραίτητα λάθος στο τελικό δίκτυο Petri που θα παραχθεί αλλά αν τα αφαιρέσουμε από το δίκτυο Petri, τότε η συμπεριφορά του μοντέλου δεν θα αλλάξει. Άρα ο επιθυμητός στόχος είναι να παραχθεί ένα δίκτυο Petri χωρίς αυτές τις 2 καταστάσεις καθώς δεν προσθέτουν καμία λειτουργικότητα και μόνο περιπλέκουν το μοντέλο.

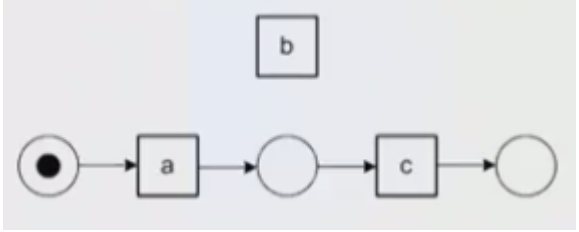


Εικόνα 23:
Οι δύο καταστάσεις p1 και p2 είναι «υπονοούμενες καταστάσεις» (**implicit places**).

Ένα δεύτερο μειονέκτημα του αλγόριθμου **Alpha** είναι ότι δεν μπορεί να αντιμετωπίσει επαναλήψεις (**loops**) συγκεκριμένου μήκους (μήκους ένα και μήκους δύο). Παράδειγμα το σύνολο γεγονότων L_2 = [$(a,c)^2$, $(a,b,c)^3$, $(a,b,b,c)^2$, $(a,b,b,b,b,c)^1$] το οποίο πάντα ξεκινάει με μια δραστηριότητα a και καταλήγει με μια δραστηριότητα c, αλλά στο ενδιάμεσο μπορεί να υπάρξει οποιοσδήποτε αριθμός από b. Αν εφαρμοστεί ο αλγόριθμος **Alpha** πάνω σε αυτό το σύνολο γεγονότων L_2 τότε θα βρεθούν οι παρακάτω σχέσεις

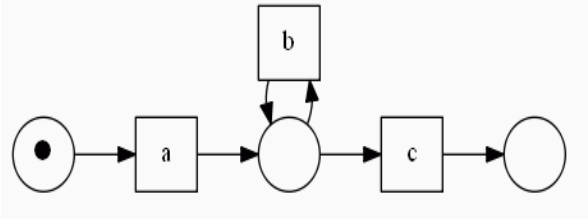
- (a>b , a>c , b>b , b>c)
- (a->b , a->c , b->c)
- (b||b)
- (a#a , c#c)

και θα βρεθεί το δίκτυο Petri της παρακάτω εικόνας 24 όπου η δραστηριότητα **b** δεν συνδέεται με καμία κατάσταση του δικτύου Petri, γιατί αν εξεταστεί με προσοχή ο αλγόριθμος **Alpha** θα δούμε ότι αν το b ακολουθεί τον εαυτό του και είναι σε σχέση παραλληλίας (**concurrency**) τότε δεν μπορεί να συνδεθεί με κανένα άλλο στοιχείο του παραγώμενου μοντέλου. Αυτό που πραγματικά έπρεπε να επιστρέψει ο αλγόριθμος είναι το δίκτυο Petri της εικόνας 25.



Εικόνα 24:
Δίκτυο Petri που παράγεται από τον αλγόριθμο **Alpha** για ένα σύνολο συμβάντων L_2. Σε

αυτήν την περίπτωση ο αλγόριθμος δεν μπορεί να ανιχνεύσει επαναλήψεις μήκους ένα.



Εικόνα 25:
Επιθυμητό αποτέλεσμα ενός αλγορίθμου εξόρυξης διαδικασιών για επαναλήψεις μήκους 1.

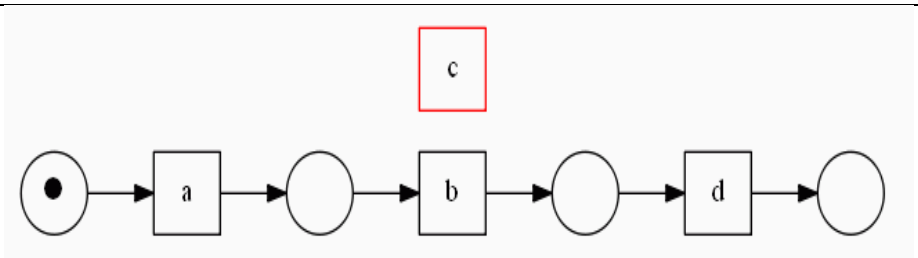
Ένα τρίτο μειονέκτημα του αλγόριθμου **Alpha** είναι ότι δεν μπορεί να ανιχνεύσει σωστά επαναλήψεις μήκους δύο. Για παράδειγμα το σύνολο γεγονότων

L_3 = [$(a, b, d)^3$, $(a, b, c, b, d)^2$, $(a, b, c, b, c, b, d)^1$]

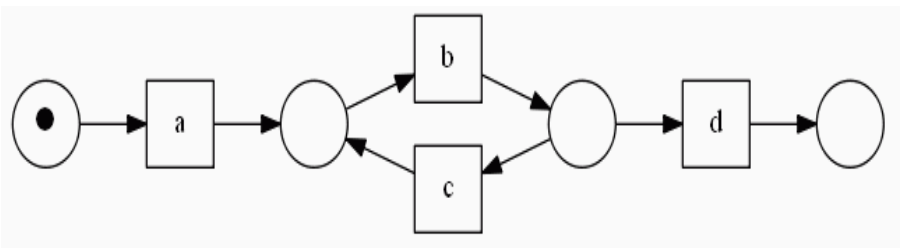
Στο σύνολο γεγονότων L_3 βλέπουμε τις μεταβάσεις b και d και ανάμεσα τους οποιοσδήποτε αριθμός από επαναλήψεις cb, cb, cb. Αν εφαρμοστεί ο αλγόριθμος **Alpha** πάνω στο σύνολο γεγονότων L_3 τότε θα βρεθούν οι παρακάτω σχέσεις

- (a>b, b>c, b>d, c>b)
- (a->b , b->d)
- (b||c)

και θα βρεθεί το δίκτυο Petri της παρακάτω εικόνας 26 όπου η μετάβαση **c** δεν συνδέεται με καμία κατάσταση του Petri Net. Όμως χρησιμοποιώντας τεχνικές όπως το pre και post processing το πρόβλημα αυτό μπορεί να ξεπεραστεί.



Εικόνα 26:
Δίκτυο Petri που παράγεται από τον αλγόριθμο **Alpha** για σύνολο γεγονότων L_3. Σε αυτήν την περίπτωση ο αλγόριθμος δεν μπορεί να ανιχνεύσει επαναλήψεις μήκους δύο.



Εικόνα 27:
Επιθυμητό αποτέλεσμα αλγορίθμου εξόρυξης διαδικασιών για επαναλήψεις μήκους 2.

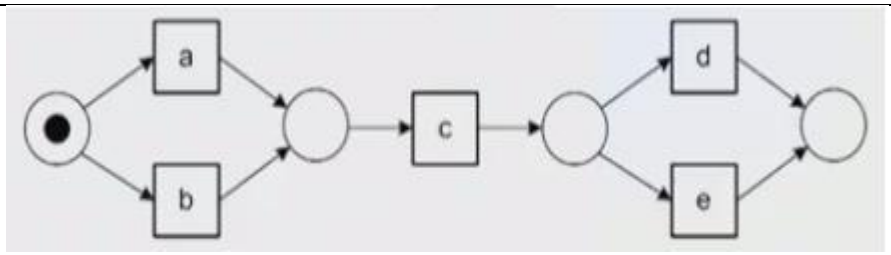
Με τα υπόλοιπα μήκη επαναλήψεων ο αλγόριθμος **Alpha** συμπεριφέρεται χωρίς κανένα πρόβλημα και παράγει τα αναμενόμενα δίκτυα Petri.

Ένα τέταρτο μειονέκτημα του αλγόριθμου **Alpha** είναι ότι δεν μπορεί να ανιχνεύσει τις «μη-τοπικές εξαρτήσεις» (**non-local dependencies)**. Ο λόγος που συμβαίνει αυτό είναι ότι τέτοιου είδους σχέσεις μεταξύ των στοιχείων του σύνολου γεγονότων δεν είναι ορατές στα δεδομένα. Αυτό βέβαια δεν είναι μόνο πρόβλημα του αλγόριθμου **Alpha,** αλλά και οποιουδήποτε άλλου αλγορίθμου εξόρυξης διαδικασιών.

Για παράδειγμα το σύνολο γεγονότων

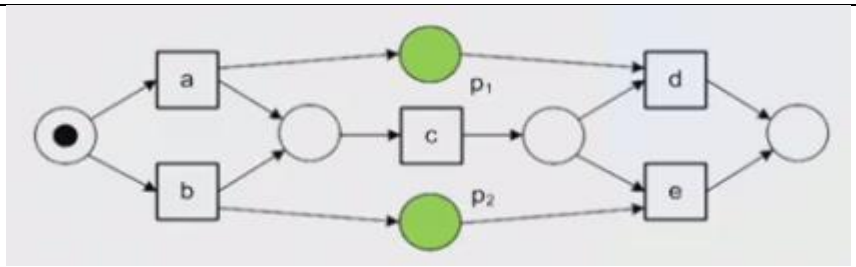
L_4 = [$(a, c, d)^{45}$, $(b, c, e)^{42}$]

το οποίο είναι πάρα πολύ απλό και βλέπουμε το a που ακολουθείται από το c το οποίο ακολουθείται από το d. Ακόμα βλέπουμε το b που ακολουθείται από το c το οποίο ακολουθείται από το e. Αν τρέξουμε τον αλγόριθμο **Alpha** θα έχουμε σαν αποτέλεσμα το μοντέλο της **Εικόνας 28** και βλέπουμε ότι τα δύο ίχνη δραστηριοτήτων που υπάρχουν στο σύνολο γεγονότων L_4 είναι όντως πιθανά. Αλλά όμως υπάρχει πιθανότητα να υπάρξουν και τα ίχνη δραστηριοτήτων bcd και ace τα οποία δεν είναι διαθέσιμα στο σύνολο γεγονότων L_4. Αυτό συμβαίνει γιατί προφανώς υπάρχει μια σχέση ανάμεσα στα a και d αλλά και στα b και e, αλλά δεν ακολουθούν το ένα το άλλο άμεσα.



Εικόνα 28

Το επιθυμητό μοντέλο που έπρεπε να παράγει ο αλγόριθμος **Alpha** είναι το μοντέλο της **εικόνας 29**, στην οποία φαίνονται οι 2 καταστάσεις p1 και p2 οι οποίες δεν βρέθηκαν από τον αλγόριθμο (μη-τοπικές εξαρτήσεις), επειδή τα a και d αλλά και b και e δεν ακολουθούν άμεσα το ένα το άλλο, αλλά μόνο έμμεσα.



Εικόνα 29:
Οι καταστάσεις p1 και p2 είναι **«**μη-τοπικές εξαρτήσεις**»** οι οποίες δεν βρέθηκαν από τον αλγόριθμο.

3. Εισαγωγή στην γλώσσα προγραμματισμού "Scala"

Στο κεφάλαιο αυτό παρουσιάζεται μια εισαγωγή στην γλώσσα προγραμματισμού Scala η οποία χρησιμοποιήθηκε για την υλοποίηση του αλγόριθμου εξόρυξης διαδικασιών Alpha.

3.1 Γλώσσα προγραμματισμού Scala

Στην συγκεκριμένη διπλωματική εργασία η υλοποίηση του αλγόριθμου **Alpha** έγινε στην γλώσσα προγραμματισμού **Scala**. Σε αυτό το κεφάλαιο θα γίνει μια σύντομη παρουσίαση των βασικών χαρακτηριστικών αυτής της γλώσσας προγραμματισμού που χρησιμοποείται ευρέως σε εφαρμογές Μεγάλων Δεδομένων σε συνδυασμό με το περιβάλλον ανάπτυξης εφαρμογών **Spark**. Πολλές σχεδιαστικές αποφάσεις της **Scala** έχουν σαν στόχο να αντιμετωπίσουν αδυναμίες της **Java**.

Η **Scala** μια γλώσσα προγραμματισμού γενικού σκοπού που παρέχει υποστήριξη για συναρτησιακό προγραμματισμό (**functional programming**) και είναι μια **strong static typing** γλώσσα. **Static typing** σημαίνει ότι ο έλεγχος του τύπου των δεδομένων (**data type**) συμβαίνει κατά την διάρκεια του compile. Ο προγραμματιστής πρέπει να ορίσει τον τύπο των μεταβλητών μέσα στον κώδικα και οποιαδήποτε ενέργεια (**operation**) γίνει στα δεδομένα θα πρέπει να ελεγχθεί από τον **compiler** για τον αν είναι δυνατή ή όχι. Το ακριβώς αντίθετο είναι οι **Dynamic typing** γλώσσες (σε αυτήν την κατηγορία **δεν** ανήκει η **Scala**) στις οποίες ο έλεγχος του τύπου δεδομένων γίνεται κατά την διάρκεια της εκτέλεσης (**runtime)** και τα σφάλματα παράγονται κατά την διάρκεια εκτέλεσης του προγράμματος αν γίνει προσπάθεια εκτέλεσης μιας μη επιτρεπτής ενέργειας (**operation**) πάνω σε μη συμβατά δεδομένα (**incompatible types**). Ακόμα όπως αναφέρθηκε και πιο πάνω η **Scala** είναι **strong typed**, που σημαίνει ότι επιτρέπονται πράξεις πάνω στα δεδομένα που έχουν συγκεκριμένο τύπο δεδομένων. Αντίθετα **weak typing** γλώσσες είναι αυτές που επιτρέπουν χειρισμό δεδομένων χωρίς να λάβουν υπόψιν τον τύπο δεδομένων.

Ο πηγαίος κώδικας (**source code**) της **Scala** πρέπει να χτιστεί σε Java bytecode, ώστε το παραγόμενο εκτελέσιμο να μπορεί να τρέξει σε ένα **JVM** (**Java Virtual machine**). Η **Scala** παρέχει γλωσσική διαλειτουργικότητα (**language interoperability)** με την Java, ώστε βιβλιοθήκες που έχουν γραφεί στην μία ή στην άλλη γλώσσα να μπορούν να γίνουν διαθέσιμες άμεσα σε κώδικα **Java** ή **Scala.** Επιπλέον όπως η C και η Java είναι μια αντικειμενοστραφής γλώσσα.

Η **Scala** διαθέτει πολλά στοιχεία του συναρτησιακού προγραμματισμού, όπως:

- **Καθαρές συναρτήσεις (Pure Functions):** είναι αυτές οι συναρτήσεις οι οποίες δεν έχουν παρενέργειες (side effects), δηλαδή δεν αποθηκεύουν κάτι στην μνήμη ή δεν εκτελούν κάποια ενέργεια εισόδου / εξόδου (I/O) αλλά και δεν έχουν εξάρτηση σε

τίποτα άλλο πέραν της εισόδου τους. Για παράδειγμα ο παρακάτω κώδικας, ο οποίος δεν επεξεργάζεται την είσοδο, που σημαίνει ότι το αποτέλεσμα είναι σταθερό ανάλογα με την λίστα των παραμέτρων.

```
def add(a:Int,b:Int) = a + b
```

Δηλαδή για κάθε είσοδο, το μόνο αποτέλεσμα είναι η έξοδος η οποία παράγει. Το αποτέλεσμα της πρόσθεσης δεν παράγει καμία παρενέργεια. Δεν αλλάζει τις τιμές των εισόδων και επιστρέφει πίσω το άθροισμα.

Μία από τις βασικές αρχές του συναρτησιακού προγραμματισμού είναι να γράφονται οι εφαρμογές ώστε ο πυρήνας της εφαρμογής να είναι γραμμένος από καθαρές συναρτήσεις, και τα οι μέθοδοι που έχουν παρενέργειες να βρίσκονται σε ένα εξωτερικό επίπεδο. Αυτό έχει σαν αποτέλεσμα ο ο κώδικας να τεστάρετε πολύ πιο εύκολα καθώς αρκεί να ελεγθεί μόνο η έξοδος της μεθόδου και όχι το αποτέλεσμα από κάποια παρενέργεια, όπως θα γινόταν σε μια μη-καθαρή συνάρτηση. Ακόμα αυτές οι μέθοδοι είναι πιο εύκολο να αποσφαλματωθούν (debug) γιατί δεν είναι απαραίτητο ο προγραμματιστής να κοιτάξει έξω από το πεδίο εφαρμογής της μεθόδου.

- **Αμετάβλητα δεδομένα (Immutable Data)**: Στο συναρτησιακό προγραμματισμό είναι απαραίτητη η χρήση αμετάβλητων δεδομένων. Τα δεδομένα αυτά δεν μπορούν να αλλαχθούν μετά την δημιουργία τους. Ο μόνος τρόπος να αλλάξουν τέτοια δεδομένα είναι να δημιουργηθεί ένα μεταβλητό αντίγραφο και να περαστούν νέα δεδομένα πάνω σε αυτό κατά την δημιουργία του. Πλεονέκτημα των αμετάβλητων δεδομένων είναι ότι δεν χρειάζεται να κλειδωθούν κατά την διάρκεια εκτέλεσης του προγράμματος, ακόμα και αν τα δεδομένα αυτά είναι απαραίτητα από πολλά νήματα (threads) καθώς κανένα δεν μπορεί να αλλάξει την τιμή τους. Έτσι αυτό βοηθάει στον συγχρονισμό (**concurrency)**. Ένα άλλο πλεονέκτημα είναι η αποθήκευση (**persistence**) των δεδομένων καθώς παλαιότερες εκδόσεις των δεδομένων μπορούν να επαναχρησιμοποιηθούν με ασφάλεια διότι δεν πρόκειται να αλλάξουν ποτέ στο μέλλον. Ακόμα δεν είναι απαραίτητο να υπάρχει ιστορικό με το πότε και με τι τιμές άλλαξαν τα δεδομένα, καθώς δεν αλλάζουν ποτέ.

- **Αντικείμενα πρώτης κατηγορίας (First class objects):** Στην **Scala** οι συναρτήσεις είναι αντικείμενα πρώτης κατηγορίας που σημαίνει ότι μπορούν να αποθηκευτούν σαν μεταβλητές (**variables**), αντικείμενα (**objects**) ή πίνακες (**arrays**). Ακόμα μπορούν να περαστούν σαν παράμετροι (**arguments**) σε μια μέθοδο αλλά και να επιστραφούν από μια μέθοδο.

- **Currying :** Είναι η τεχνική που επιτρέπει την μετατροπή μιας συνάρτησης με πολλά ορίσματα (**arguments**) σε μια συνάρτηση με ένα μόνο όρισμα. Το μόνο όρισμα είναι η τιμή του πρώτου ορίσματος από την αρχική συνάρτηση η οποία επιστρέφει μια

άλλη μέθοδο με ένα όρισμα. Αυτή η επιστρεφόμενη μέθοδος με την σειρά της θα πάρει σαν όρισμα το δεύτερο όρισμα της αρχικής μεθόδου και θα επιστρέψει μια άλλη μέθοδο με ένα όρισμα. Αυτό το δέσιμο μεταξύ των μεθόδων συμβαίνει για τον αριθμό των ορισμάτων της αρχικής μεθόδου. Το τελευταίο στην αλυσίδα θα έχει πρόσβαση σε όλα τα ορίσματα και θα μπορεί να τα χειριστεί όπως θέλει. Παρακάτω φαίνεται ένα παράδειγμα υλοποίησης της τεχνικής **currying** στην **Scala**.

```
def add(x: Int): (Int => Int) = {
  (y: Int) => {
    x + y
  }
}
```

Εικόνα 30:
Αυτή η μέθοδος δέχεται ένα όρισμα και επιστρέφει πίσω μια μέθοδο η οποία εκτελεί άθροιση. Η εκτέλεση της γίνεται με "**add(1).apply(1)**" και επιστρέφει σαν αποτέλεσμα 2 (το αναμενόμενο αποτέλεσμα της άθροισης).

Επειδή η **Scala** υποστηρίζει **curried** συναρτήσεις, ο κώδικας της **Εικόνας 30,** μπορεί να μετατραπεί σε μια πιο απλή **curried** έκδοση απλά ξεχωρίζοντας τις παραμέτρους (**Εικόνα 31**).

```
def add(x: Int)(y: Int): Int = {
  x + y
}
```

Εικόνα 31:
Αυτή η μέθοδος εκτελείται με δύο τρόπους

- (add(1) _).apply(1) με αποτέλεσμα 2
- add(1)(1) με αποτέλεσμα 2

- **Συμπέρασμα τύπου δεδομένων (Type inference):** Αναφέρεται στον αυτόματο εντοπισμό ένος τύπου δεδομένων σε μια γλώσσα προγραμματισμού. Είναι ένα χαρακτηριστικό που υπάρχει σε μερικές **strongly statically typed** γλώσσες. Επιπλέον είναι ένα χαρακτηριστικό των συναρτησιακών γλωσσών προγραμματισμού. Ορισμένες γλώσσες που έχουν αυτό το χαρακτηριστικό είναι η C++11, C#, Go, Haskell, Java(από την έκδοση 10), Kotlin, Scala.
  Στην Scala συγκεκριμένα ο compiler μπορεί να καταλάβει τον τύπο δεδομένων χωρίς να έχει οριστεί ξεκάθαρα από τον προγραμματιστή. Στο παρακάτω παράδειγμα ο compiler μπορεί να καταλάβει ότι το **businessName** είναι τύπου String

  ```
  val businessName = "Montreux Jazz Café"
  ```

  Ακόμα στο επόμενο παράδειγμα ο compiler βλέπει το τύπο δεδομένου της επιστρεφόμενης τιμής της μεθόδου σαν Integer και έτσι δεν χρειάζεται να δηλωθεί συγκεκριμένα ο επιστρεφόμενος τύπος δεδομένου (data type).

```
def squareOf(x: Int) = x * x
```

- **Καθυστερημένη Εκτίμηση (Lazy evaluation)**: Είναι μια τεχνική εκτίμησης η οποία καθυστερεί τον υπολογισμό μιας έκφρασης έως ότου η τιμή της έκφρασης αυτής χρειάζεται πραγματικά. Σήμερα οι περισσότερες μοντέρνες γλώσσες προγραμματισμού υποστηρίζουν την τεχνική της καθυστερημένης εκτίμησης. Σε αντίθεση με την άμεση εκτίμηση (**eager** ή **strict evaluation**), η οποία υπολογίζει τιμές το συντομότερο δυνατό, η καθυστερημένη εκτίμηση παρουσιάζει πλεονεκτήματα όπως ότι μπορεί να βελτιώσει την απόδοση της εφαρμογής επειδή δεν θα εκτελέσει κανέναν υπολογισμό μέχρι αυτός να χρειαστεί πραγματικά. Αυτό είναι πολύ θετικό γιατί μπορεί να υπάρχουν υπολογισμοί σε μια εφαρμογή οι οποίοι τελικά δεν θα εκτελεστούν ποτέ, οπότε με την καθυστερημένη εκτίμηση γίνεται οικονομία σε υπολογιστική ισχύ. Ακόμα μπορεί να αυξηθεί ο χρόνος απόκρισης μιας εφαρμογής αναβάλοντας βαριές λειτουργίες έως ότου είναι απολύτως απαραίτητες. Από την άλλη πλευρά το μειονέκτημα είναι ότι η απόδοση (**performance**) δεν είναι προβλέψιμη επειδή δεν είναι γνωστό το πότε θα χρειαστεί να γίνει να εκτιμηθεί η τιμή αυτή.
  Η **Scala** μπορεί να κάνει χρήση της καθυστερημένης εκτίμησης χρησιμοποιώντας τα **lazy vals**. Μια μεταβλητή στην **Scala** μπορεί να χαρακτηριστεί σαν **lazy** (δηλαδή η τιμή της να υπολογιστεί μόλις γίνει η κλήση της μεταβλητής), χρησιμοποιώντας απλά την λέξη κλειδί **lazy**. Για παράδειγμα

  ```
  lazy val lval = 10
  ```

  Άλλος τρόπος για να κάνει χρήση της καθυστερημένης εκτίμησης η **Scala** είναι να χρησιμοποιεί **call by name** παραμέτρους. Η τιμή της παραμέτρου θα υπολογιστεί μόνο όταν πρόκειται να χρησιμοποιηθεί μέσα στην μέθοδο. Για παράδειγμα η παρακάτω υπογραφή (signature) της μεθόδου έχει μια **call by name** παράμετρο.

  ```
  def method(n :=> Int)
  ```

3.2 Λόγοι για να ασχοληθεί κάποιος με την Scala

Υπάρχουν αρκετοί λόγοι για να μάθει κάποιος προγραμματιστής την γλώσσα προγραμματισμού **Scala**. Τα τελευταία χρόνια η **Scala** έχει μετατραπεί στην καλύτερη εναλλακτική λύση της Java, καθώς είναι και αυτή μια **JVM** (Java Virtual Machine) γλώσσα, η οποία έχει αφήσει αρκετά πίσω άλλες **JVM** γλώσσες όπως η **Groovy** και η **Clojure**.

Η **Scala** υποστηρίζει δύο παραδείγματα προγραμματισμού. Τον αντικειμενοστραφή προγραμματισμό (**OOP**) αλλά και τον συναρτησιακό προγραμματισμό (**functional programming**). Για κάποιον προγραμματιστή ο οποίος επιθυμεί να διευρύνει τις γνώσεις του στην επιστήμη των υπολογιστών είναι καλό να γνωρίζει τουλάχιστον μια γλώσσα από κάθε παράδειγμα προγραμματισμού (επιτακτικό, συναρτησιακό, αντικειμενοστραφή και λογικό) και η **Scala** παρέχει την δυνατότητα της συγγραφής κώδικα πάνω σε δύο από αυτά. Ο συνδιασμός αυτών των δύο χαρακτηριστικών κάνει δυνατή την συγγραφή προγραμμάτων τα οποία είναι καθαρογραμμένα (**clean code**) και ευανάγνωστα (**readable**).

Το να χρησιμοποιεί δύο παραδείγματα προγραμματισμού (συναρτησιακό και αντικειμενοστραφή) η **Scala,** είναι ένα από τα δυνατά χαρτιά της γλώσσας το οποίο ακολούθησε και η **Java** με την έκδοση 8 με την εισαγωγή των εκφράσεων **lamda**.

Ένα από τα πλεονεκτήματα της **Scala** είναι ότι υποστηρίζει συμβατότητα με την **Java.** Αυτό σημαίνει ότι ο κώδικας της **Scala** μπορεί να χρησιμοποιήσει βιβλιοθήκες της **Java** απευθείας χωρίς καμία παραμετροποίηση. Ακόμα είναι δυνατόν και το αντίθετο, δηλαδή να καλέσει ο προγραμματιστής από **Java** κώδικα, κώδικα **Scala.** Έτσι είναι δυνατόν κάποιος να κρατήσει τα υπάρχοντα προγράμματα σε **Java** και να γράψει τα καινούρια σε **Scala** πάρα πολύ εύκολα.

Η **Scala** ακολουθεί πολλές καλές πρακτικές (**best practices**) και μοτίβα τα οποία είναι ενσωματωμένα στην βασική έκδοση της γλώσσας. Σκοπός της **Scala** ήταν να εφαρμόσει πολλές πρόσφατες καινοτομίες σε μια γλώσσα ώστε να χρησιμοποιηθεί όσο το δυνατόν περισσότερο από τους προγραμματιστές, όπως ακριβώς και η **Java**.

Πολλές εταιρείες τα τελευταία χρόνια σκέφτονται να χρησιμοποιήσουν ή χρησιμοποιούν ήδη **Scala**. Παραδείγματα τέτοιων εταιρειών είναι το Twitter, η Linkedin, το Foursquare και το Quora. Έτσι όπως γίνεται κατανοητό η γνώση της **Scala** θα κάνει κάποιον προγραμματιστή ανταγωνιστικότερο στην αγορά εργασίας. Ακόμα υπάρχουν πολλά αναπτυσσόμενα περιβάλλοντα ανάπτυξης (**frameworks**) όπως το **Akka**, το **Play** και το **Spark**. Το **Akka** είναι ένα βασιζόμενο σε **Scala** περιβάλλον ανάπτυξης το οποίο χρησιμοποιείται για την δημιουργία παράλληλων, κατανεμημένων και οδηγούμενων από γεγονότα (**concurrent**, **distributed event-driven**) εφαρμογών σε **JVM**. Ακόμα η **Scala** έχει χρησιμοποιηθεί και στην κατηγορία των Μεγάλων Δεδομένων καθώς πάνω σε αυτή τη γλώσσα έχει γραφεί το **Spark,** ένα περιβάλλον ανάπτυξης για επεξεργασία δεδομένων σε μια συστάδα υπολογιστών (**cluster)**. Επιπλέον όλο και περισσότεροι προγραμματιστές ασχολούνται πλέον με την **Scala** και αυτό έχει σαν επακόλουθο όλο και περισσότερα περιβάλλοντα ανάπτυξης και βιβλιοθήκες να χτίζονται πάνω στην **Scala**. Όλο και περισσότερα περιβάλλοντα συγγραφής κώδικα (IDEs) υποστηρίζουν την σύνταξη της **Scala**, όπως το **Eclipse** και το **IntelliJ** αλλά και πολλά build εργαλεία (όπως το SBT, Maven και Ant).

Τέλος, για κάποιον προγραμματιστή **Java**, το να μάθει κάποια συναρτησιακή γλώσσα προγραμματισμού όπως η Haskell ή η OCaml είναι αρκετά δύσκολο. Σε αντίθεση με την **Scala** που μπορεί να μάθει σχετικά εύκολα καθώς είναι μια γλώσσα που βασίζεται στον αντικειμενοστραφή προγραμματισμό και έχει αρκετά εύκολη σύνταξη με πολλά κοινά στοιχεία με την **Java**.

4. Περιβάλλον Ανάπτυξης Εφαρμογών Spark

Στο κεφάλαιο αυτό παρουσιάζεται μια εισαγωγή στο περιβάλλον ανάπτυξης εφαρογών Spark, στα δομικά του στοιχεία αλλά και στην ροή εκτέλεσης ενός προγράμματος Spark. Ακόμα παρουσιάζονται οι διάφορες δομές δεδομένων του Spark που κατανέμονται στους κόμβους μιας συστάδας υπολογιστών αλλά και οι κοινόχρηστες μεταβλητές για όλους τους κόμβους της συστάδας υπολογιστών.

4.1 Εισαγωγή στο Περιβάλλον Ανάπτυξης Εφαρμογών Spark

Το **Apache Spark** συνήθως ορίζεται σαν μια γρήγορη, γενικού σκοπού, κατανεμημένης εργασίας πλατφόρμα (**distributed computing platform**). Το **Apache Spark** έφερε επανάσταση στον χώρο των Μεγάλων Δεδομένων επειδή κάνει αποδοτική χρήση της μνήμης και μπορεί να εκτελέσει διάφορες εργασίες (**jobs**) 10 έως 100 φορές γρηρότερα από το **Hadoop's MapReduce**. Επιπλέον οι δημιουργοί του **Apache Spark** κατάφεραν να κάνουν αφαιρέσουν το πως ο προγραμματιστής χειρίζεται τα μηχανήματα (**machines**) μιας συστάδας υπολογιστών και αντί αυτού να παρουσιάζουν ένα σύνολο από βασισμένα σε σύνολα (**collections**) **APIs**. Η εργασία με τα σύνολα του **Spark** είναι παρεμφερής με την εργασία με τα σύνολα της **Scala**, της **Java** ή της **Python**, αλλά τα σύνολα του **Spark** κατανέμονται σε πολλά μηχανήματα (**nodes**) της συστάδας. Η επεξεργασία αυτών των δεδομένων μεταφράζονται σε πολύπλοκα παράλληλα προγράμματα χωρίς να το αντιλαμβάνεται απαραίτητα αυτό ο χρήστης, κάτι το οποίο είναι πολύ δυνατό χαρακτηριστικό του **Spark** καθώς απλοποιεί την διαδικασία συγγραφής **Spark** προγραμμάτων.

Το **Apache Spark** είναι μια πολύ ενδιαφέρουσα νέα τεχνολογία που συνεχώς ξεπερνάει το **Hadoop's MapReduce** σαν την προτεινόμενη πλαφόρμα για την επεξεργασία Μεγάλων Δεδομένων. Το **Hadoop** είναι ένα ανοιχτού κώδικα, κατανεμημένο (**distributed**) περιβάλλον υλοποίησης υπολογισμού δεδομένων γραμμένο σε **Java** και αποτελείται από το κατανεμημένο σύστημα αρχείων του **Hadoop** (**HDFS**) και το **MapReduce** (τον μηχανισμό εκτέλεσης εντολών). Το **Apache Spark** μοιάζει με το **Hadoop** στο ότι είναι μία κατανεμημένη (**distributed**), γενικού σκοπού (**general purpose**) υπολογιστική πλατφόρμα (**computing platform**). Αλλά η μοναδική σχεδίαση του **Apache Spark,** η οποία δίνει την δυνατότητα για αποθήκευση μεγάλου όγκου δεδομένων στην κύρια μνήμη, προσφέρει πάρα πολύ μεγάλη αύξηση της απόδοσης. Τα προγράμματα που τρέχουν σε **Spark** μπορεί να είναι έως και 100 φορές γρηγορότερα από αντίστοιχα προγράμματα γραμμένα με την τεχνική του **MapReduce**.

Αν και το **Apache Spark** είναι περιβάλλον υλοποίησης ανοικτού κώδικα, η **Databricks** είναι ο κύριος παράγοντας πίσω από το **Apache Spark** συνεισφέροντας πάνω από το 75% του

κώδικα. Ακόμα προσφέρει και το **Databricks Cloud**, ένα εμπορικό προιόν για ανάλυση Μεγάλων Δεδομένων που βασίζεται πάνω στο **Apache Spark**.

Χρησιμοποιώντας το API και την αρχιτεκτονική του **Apache Spark**, ο προγραμματιστής μπορεί να γράψει κατανεμημένα προγράμματα (**distributed programs**) με τρόπο σαν να έγραφε ένα πρόγραμμα για να τρέξει σε ένα μόνο μηχάνημα. Οι δομές δεδομένων (**collections**) του κρύβουν το γεγονός ότι τα δεδομένα κατανέμονται σε ένα μεγάλο αριθμό από μηχανήματα (**nodes**) σε μια συστάδα υπολογιστών. Επιπλέον το **Apache Spark** επιτρέπει την χρήση μεθόδων που βασίζονται στον συναρτησιακό προγραμματισμό, κάτι το οποίο ταιριάζει πολύ σε εργασίες επεξεργασίας δεδομένων.

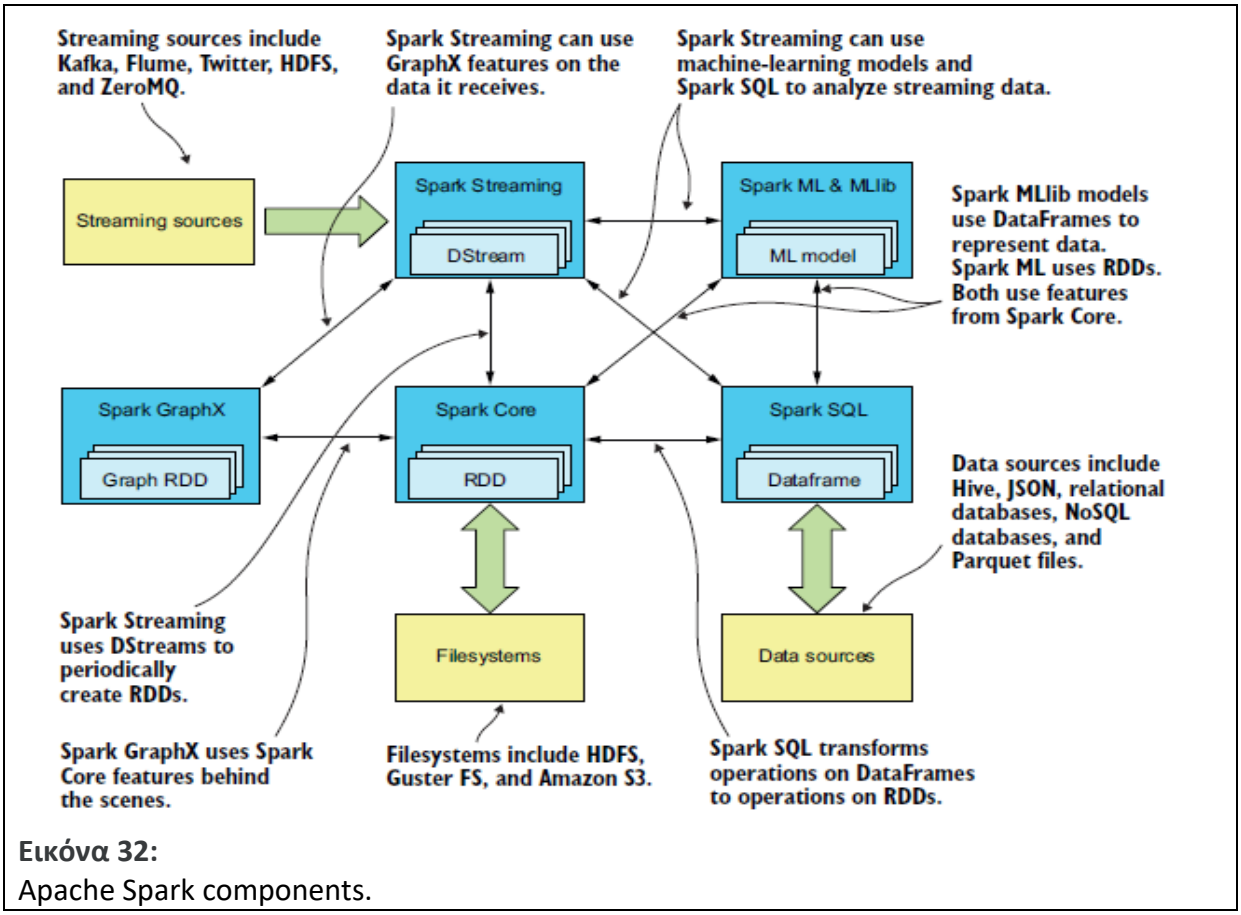
Το **Apache Spark** υποστηρίζει μια πληθώρα γλωσσών προγραμματισμού στις οποίες ο προγραμματιστής μπορεί να γράψει προγράμματα επεξεργασίας Μεγάλων Δεδομένων. Παραδείγματα είναι η **Python**, η **Scala**, η **Java** και η **R**, κάνοντας έτσι το **Apache Spark** εύκολα διαθέσιμο προς ένα μεγάλο εύρος επαγγελματιών. Συνήθως άνθρωποι από τον χώρο των επιστημών προτιμούν την **Python** και την **R,** ενώ προγραμματιστές που ασχολούνται ήδη με **Java** θα προτιμήσουν την **Java** ή την **Scala,** η οποία συνδιάζει αντικειμενοστραφές και συναρτησιακό προγραμματισμό πάνω σε ένα **Java Virtual Machine** (**JVM**).

Το **Apache Spark** συνδιάζει ικανότητες που μοιάζουν με αυτές του **Hadoop's MapReduce** για επεξεργασία παρτίδων (**batch processing**), συναρτήσεις επεξεργασίας δεδομένων σε πραγματικό χρόνο (**real-time data processing functions**), χειρισμό δομημένων δεδομένων (**structured data**) με τεχνικές **SQL**, αλγόριθμοι γράφων (**graph algorithms**) και μηχανικής μάθησης όλα σε ένα μοναδικό περιβάλλον ανάπτυξης. Έτσι το **Apache Spark** μπορεί να καλύψει τις περισσότερες ανάγκες ενός μηχανικού Μεγάλου Όγκου Δεδομένων και αυτός είναι και ο λόγος που το **Apache Spark** έχει γίνει ένα από τα πιο σημαντικά projects της **Apache Software Foundation** σήμερα.

Παρ' όλα αυτά ορισμένες εφαρμογές δεν είναι κατάλληλες για το το **Apache Spark**. Επειδή το **Apache Spark** διαθέτει μια καταναμημένη αρχιτεκτονική (**distributed architecture**), αυτό έχει σαν επακόλουθο ένα επιπλέον βάρος (**overhead**) στον χρόνο υπολογισμού (**processing time**). Αυτό το επιπλέον βάρος είναι αμελητέο για το χειρισμό πολύ μεγάλου όγκου δεδομένων, αλλά αν το σύνολο δεδομένων που διαθέτει ο προγραμματιστής είναι τέτοιο ώστε να μπορεί να επεξεργαστεί και από ένα μόνο μηχάνημα (**single machine**), πράγμα που συμβαίνει όλο και περισσότερο στις μέρες μας, μπορεί να είναι πιο αποδοτικό να χρησιμοποιηθεί κάποιο άλλο framework το οποίο είναι καταλληλότερο για τέτοιου είδους επεξεργασία δεδομένων.

4.2 Δομικά Στοιχεία του Apache Spark

Το **Apache Spark** αποτελείται από διάφορα δομικά στοιχεία (**components**) συγκεκριμένου σκοπού. Αυτά είναι τα **Spark Core**, **Spark SQL, Spark Streaming, Spark GraphX** και **Spark MLlib**. Αυτά τα δομικά στοιχεία φαίνονται στην **Εικόνα 32**. Αυτά τα δομικά στοιχεία κάνουν το **Spark** μια ενοποιημένη πλατφόρμα από διάφορες λειτουργικότητες που μπορούν να χρησιμοποιηθούν από πολλές εργασίες που προηγουμένως έπρεπε να εκπληρωθούν από διαφορετικά περιβάλλοντα ανάπτυξης.



Εικόνα 32:
Apache Spark components.

Ακολουθεί σύντομη περιγραφή του κάθε δομικού στοιχείου του **Apache Spark**.

- **Spark core**: Περιέχει την βασική λειτουργικότητα του **Apache Spark** η οποία είναι απαραίτητη για την εκτέλεση διάφορων εργασιών στο **Spark**, και χρειάζεται για την λειτουργία και άλλων δομικών στοιχείων του **Spark**. Το πιο σημαντικό στοιχείο του **Spark core** είναι το **resilient distributed dataset** (**RDD**) τα οποία είναι και το κεντρικό στοιχείο του **Spark API**. Αποτελεί αφαίρεση μιας κατανεμημένης δομής δεδομένων (**distributed collection of data**) με λειτουργίες (**operations**) και μετασχηματισμούς (**transformations**) που είναι εφαρμόσιμες πάνω σε ένα σύνολο

δεδομένων. Είναι **resilient** επειδή το **RDD** είναι ικανό να ξανά χτίσει το σύνολο δεδομένων σε περίπτωση που ο κόμβος (**node**) της συστάδας υπολογιστών αποτύχει να λειτουργήσει σωστά για οποιοδήποτε λόγο. Το **Spark core** περιέχει λογική ώστε να μπορεί να προσπελαύνει διάφορα συστήματα αρχείων, όπως το HDFS, GlusterFS, Amazon S3 και άλλα. Ακόμα παρέχει διάφορους τρόπους διαμοιρασμού πληροφορίας ανάμεσα σε κόμβους (**nodes**) μιας συστάδας υπολογιστών με τις **broadcast** μεταβλητές (**broadcast variables**) και τους συσσωρευτές (**accumulators**). Ακόμα πολλές βασικές λειτουργίες του **Spark**, όπως η ασφάλεια (**security**), η δικτύωση (**networking**), η χρονοδρονολόγηση (**scheduling**), και η ανάμειξη δεδομένων (**data shuffling**) είναι επίσης τμήμα του **Spark core**.

- **Spark SQL**: Περιέχει συναρτήσεις για την επεξεργασία μεγάλου όγκου δομημένων (**structured**) και κατανεμημένων δεδομένων χρησιμοποιώντας εντολές SQL που υποστηρίζονται από το **Spark** και Hive SQL (**HiveQL**). Με την εισαγωγή των **DataFrames** στην έκδοση 1.3 του **Spark** και των **DataSets** στην έκδοση 1.6 του **Spark**, ο χειρισμός των δομημένων δεδομένων απλουστεύεται και έτσι γίνεται δυνατή η τεράστια αύξηση της απόδοσης των **Spark** προγραμμάτων. Έτσι το **Spark SQL** έγινε ένα από τα σημαντικότερα δομικά στοιχεία του **Spark**. Ακόμα το **Spark SQL** μπορεί επίσης να χρησιμοποιείται για την ανάγνωση και την εγγραφή δεδομένων από και προς διάφορες δομημένες μορφές (**structured format**) και πηγές δεδομένων (**data sources**), όπως τα αρχεία JSON (JavaScript Object Notation), τα **Parquet** αρχεία (όλο και περισσότερο δημοφιλής μορφή αρχείου που επιτρέπει την αποθήκευση ενός σχήματος-**schema** μαζί με τα δεδομένα), σχεσιακές βάσεις δεδομένων (**relational databases**), βάσεις δεδομένων **Hive** και άλλα. Οι λειτουργίες (**operations**) πάνω στα DataFrames και στα DataSets σε κάποιο σημείο μεταφράζονται σε λειτουργίες (**operations**) πάνω στα RDDs και εκτελούνται ως κανονικές **Spark** εργασίες (**Spark jobs**).

- **Spark Streaming**: Είναι ένα δομικό στοιχείο του **Spark** για την επεξεργασία δεδομένων ροής σε πραγματικό χρόνο (**real-time streaming data**) από διάφορες πηγές. Οι υποστηριζόμενες πηγές ροής (**streaming**) περιλαμβάνουν HDFS, Kafka, Flume, Twitter, ZeroMQ και διάφορες άλλες τεχνολογίες. Οι **Spark Streaming** λειτουργίες (**operations**) σε περίπτωση αποτυχίας της ορθής λειτουργίας τους μπορούν και επανέρχονται αμέσως στην φυσιολογική χωρίς προβλήματα λειτουργία, κάτι το οποίο είναι σημαντικό για την επεξεργασία δεδομένων σε πραγματικό χρόνο (**real-time**). Το **Spark Streaming** αναπαριστά δεδομένα ροής (**streaming data**), χρησιμοποιώντας διακριτές ροές (**discretized streams** ή αλλιώς **DStreams**), οι οποίες περιοδικά δημιουργούν **RDDs** που περιέχουν δεδομένα που εισήχθησαν κατά το τελευταίο χρονικό παράθυρο (**time window**). Το **Spark Streaming** μπορεί να συνδιαστεί με άλλα δομικά στοιχεία του **Spark** μέσα σε ένα πρόγραμμα, ενοποιώντας έτσι την επεξεργασία δεδομένων σε πραγματικό χρόνο

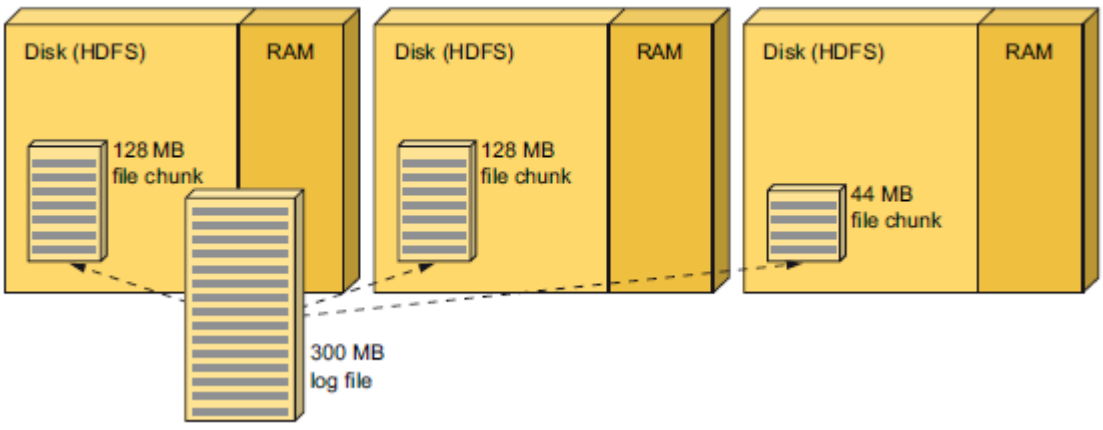
(**real-time processing**) με την μηχανική μάθηση (**machine learning**), την SQL αλλά και τις λειτουργίες πάνω σε γράφους (**graph operations**).

- **Spark Mllib:** Είναι μια βιβλιοθήκη αλγορίθμων μηχανικής μάθησης που αναπτύχθηκε από το έργο MLbase στο UC Berkeley. Οι υποστηριζόμενοι αλγόριθμοι περιλαμβάνουν την ταξινόμηση Bayes (**Bayes classification**), δέντρα αποφάσεων (**decision trees**), την γραμμική παλινδρόμηση (**linear regression**) και την ομαδοποίηση k-means (**k-means clustering**). Το **Spark Mllib** χειρίζεται μοντέλα μηχανικής μάθησης που χρειάζονται για να μετασχηματίζουν σύνολα δεδομένων, τα οποία αναπαρίστανται από RDDs ή DataFrames.

- **Spark GraphX:** Τα γραφήματα είναι δομές δεδομένων που περιλαμβάνουν τις κορυφές και τις ακμές που τις συνδέουν. Το **Spark GraphX** παρέχει συναρτήσεις για την κατασκευή γράφων, οι οποίες παρουσιάζονται ως **RDDs** σε μορφή γραφημάτων (δηλαδή **EdgeRDD** και **VertexRDD**). Το **Spark GraphX** περιέχει υλοποιήσεις των πιο σημαντικών αλγορίθμων της θεωρείας γράφων, όπως connected components, συντομότερα μονοπάτια (shortest paths) και πολλούς άλλους.

4.3 Ροή Προγράμματος Apache Spark

Σε αυτό το κεφάλαιο θα γίνει παρουσίαση ενός απλού προγράμματος **Spark**. Υποθέτουμε ότι ένα 300 mb αρχείο εγγραφών είναι αποθηκευμένο σε μια HDFS συστάδα υπολογιστών με τρεις κόμβους. Το HDFS (Hadoop Distributed File System) αποτελεί ένα σύστημα για κατανεμημένη αποθήκευση αρχείων βασιζόμενη στο Hadoop. Το HDFS αυτόματα χωρίζει το αρχείο σε τμήματα των 128 mb (σε block στην ορολογία του Hadoop) και τοποθετεί το κάθε τμήμα σε ένα ξεχωριστό κόμβο της συστάδας υπολογιστών (**Εικόνα 33**). Υποθέτουμε ότι το Spark τρέχει στο YARN, μέσα στην ίδια συστάδα υπολογιστών τύπου Hadoop.

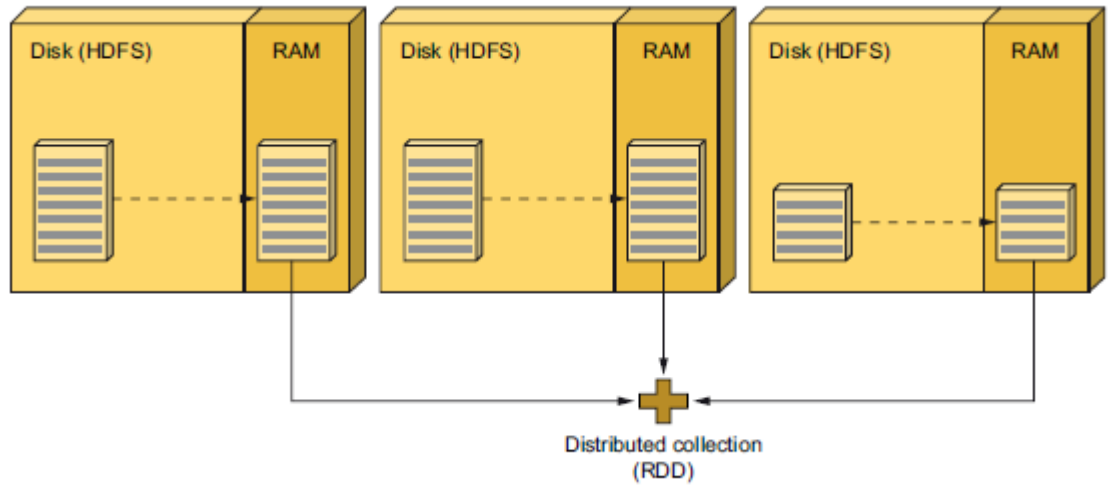


Εικόνα 33:
Αποθήκευση ενός αρχείου 300mb σε μια συστάδα υπολογιστών Hadoop με 3 κόμβους.

Σε έναν Spark μηχανικό λογισμικού ανατίθεται το πρόβλημα να αναλύσει τα δεδομένα αυτά και να βρει πόσα σφάλματα του τύπου OutOfMemory συνέβησαν τις τελευταίες δύο εβδομάδες. Υποθέτουμε ότι τα logs για τις τελευταίες δύο εβδομάδες είναι αποθηκευμένα στην συστάδα υπολογιστών Hadoop με 3 κόμβους που αναφέρθηκε πιο πάνω. Το πρόγραμμα αυτό αποτελείται από τις τρεις παρακάτω γραμμές σε Scala

```
val lines = sc.textFile("hdfs://path/to/the/file")
val oomLines = lines.filter(l => l.contains("OutOfMemoryError")).cache()
val result = oomLines.count()
```

Στην γραμμή (1) γίνεται η φόρτωση του αρχείου με τις εγγραφές από τον HDFS χρησιμοποιώντας μόνο μια γραμμή κώδικα **Scala**. Αυτό φαίνεται και στην **Εικόνα 34** όπου τα δεδομένα μεταφέρονται στην μνήμη RAM κάθε κόμβου της συστάδας υπολογιστών. Τώρα το **Spark** έχει ένα δείκτη προς κάθε **partition** των δεδομένων στην μνήμη RAM. Το άθροισμα των **partitions** είναι μια κατανεμημένη συλλογή των γραμμών ενός αρχείου εγγραφών το οποίο απεικονίζεται από ένα **RDD** που έχει αρχικοποιηθεί στην πρώτη γραμμή του κώδικα. Με λίγα λόγια, μπορούμε να πούμε ότι τα **RDDs** επιτρέπουν να δουλέψει κάποιος με μια κατανεμημένη δομή δεδομένων (**distributed data collection**) με τον ίδιο ακριβώς τρόπο που θα δούλευε με οποιαδήποτε δομή δεδομένων η οποία δεν είναι κατανεμημένη σε κάποια συστάδα υπολογιστών αλλά βρίσκεται αποθηκευμένη σε ένα μηχάνημα (local machine). Επιπλέον ο προγραμματιστής δεν χρειάζεται να χειριστεί θέματα που προκύπτουν επειδή τα δεδομένα είναι κατανεμημένα (**distributed**) σε πολλά μηχανήματα ούτε και διάφορα προβλήματα που μπορεί να προκύψουν σε ορισμένα μηχανήματα της συστάδας.



Εικόνα 34:
Φόρτωση ενός αρχείου κειμένου από τον HDFS.

Επιπλέον το RDD παρέχει και ένα API, που επιτρέπει στον προγραμματιστή να δουλεύει με μια συλλογή δεδομένων με συναρτησιακή λογική. Για παράδειγμα είναι δυνατό το φιλτράρισμα των δεδομένων με την εντολή **filter**, η μετατροπή σε κάποιον άλλον τύπο

δεδομένων χρησιμοποιώντας την εντολή **map** και μια συνάρτηση (**function**) και το **reduction** των δεδομένων σε μια **cumulative** τιμή με την εντολή reduce. Ακόμα είναι δυνατή η αφαίρεση (**subtract**), η τομή (**intersect**) και η δημιουργία ένωσης (**union**) σε σχέση με κάποιο άλλο RDD.

Έτσι λοιπόν έχοντας ήδη μια απεικόνηση του RDD το οποίο έχει δημιουργηθεί από την γραμμή 1 του παραπάνω κώδικα, για να βρεθεί ο αιρθμός των OutOfMemory σφαλμάτων που συνέβησαν τις τελευταίες δύο εβδομάδες, πρώτα πρέπει να αφαιρεθούν όλες οι γραμμές οι οποίες δεν περιέχουν το substring OutOfMemory. Αυτό μπορεί να γίνει χρησιμοποιώντας τον **τελεστή** filter όπως φαίνεται στην γραμμή 2 του κώδικα. Με την κλήση της μεθόδου **cache()** στην ίδια γραμμή δίνεται η εντολή στο **Spark** να αφήσει το RDD στην μνήμη για όλες τις εκτελέσεις. Έτσι εκτελώντας την εντολή στην τρίτη γραμμή το **Spark** υπολογίζει τον αριθμό των OutOfMemory σφαλμάτων που συνέβησαν τις τελευταίες δύο εβδομάδες.

Σύμφωνα με τα παραπάνω, βλέπουμε ότι το Spark επιτρέπει την εκτέλεση filtering και counting κατανεμημένων δεδομένων (**distributed data**) μόλις σε τρεις γραμμές κώδικα εκτελώντας τον κώδικα σε τρεις κόμβους παράλληλα. Αν ο προγραμματιστής θελήσει να καλέσει ένα επιπλέον filtering στα δεδομένα, το **Spark** δεν θα ξανά φορτώσει το αρχείο από το **HDFS** αλλά θα το φορτώσει από την cache, καθώς αποθηκεύτηκε εκεί στην τελευταία γραμμή του κώδικα.

Το παραπάνω παράδειγμα δείχνει ότι τα RDDs αποτελούν μια αφαίρεση σε κατανεμημένα δεδομένα (**distributed collections**) και ότι ο χειρισμός και η ανάλυση μεγάλου όγκου δεδομένων μπορεί να γίνει παράλληλα με σχετικά απλή υλοποίηση.

4.4 Αρχιτεκτονική του Apache Spark

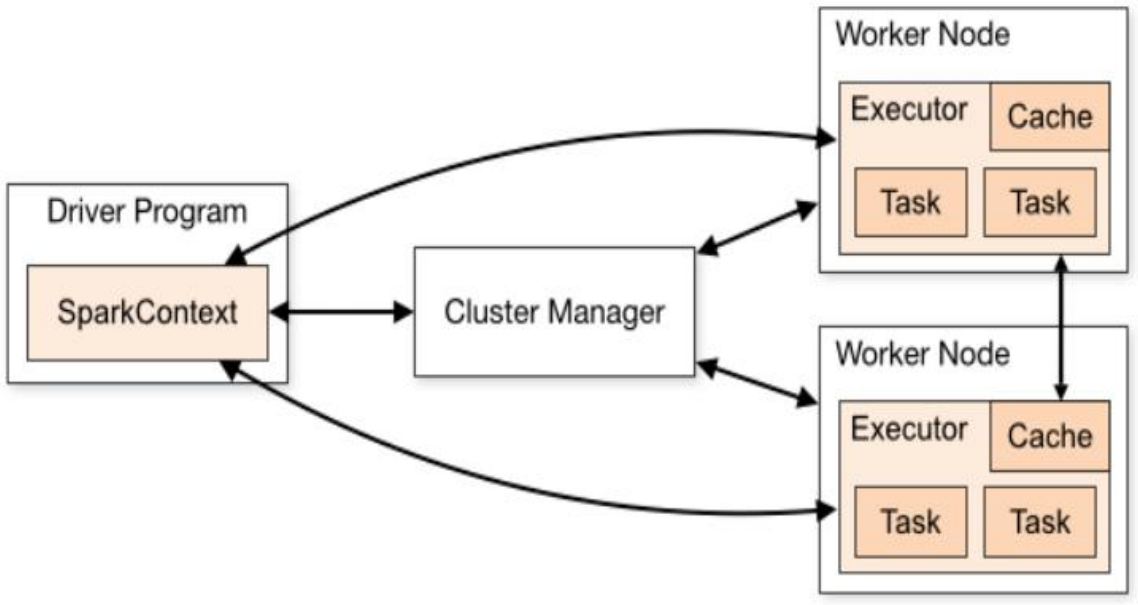
Το περιβάλλον ανάπτυξης **Apache Spark** έχει μια καλά ορισμένη και σε στρώσεις αρχιτεκτονική όπου όλα τα δομικά στοιχεία του **Spark** κα τα **Spark layers** είναι χαλαρά συνδεδεμένα και αλληλεπιδρούν με πολλές βιβλιοθήκες και διάφορες επεκτάσεις (**extentions**). Η αρχιτεκτονική του **Apache Spark** βασίζεται σε δύο βασικές αφαιρέσεις οι οποίες είναι οι παρακάτω

- **Resilient Distributed Datasets (RDD)**. Είναι δεδομένα που χωρίζονται σε partitions και αποθηκεύονται στην μνήμη των **worker** κόμβων σε μια συστάδα υπολογιστών. Τα **RDDs** υποστηρίζουν δύο διαφορετικούς τύπους λειτουργιών, τους μετασχηματισμούς (**transformations**) και τις ενέργεις (**actions**).

- **Directed Acyclic Graph (DAG). DAG** είναι μια ακολουθία από υπολογισμούς που εφαρμόζεται πάνω στα δεδομένα όπου κάθε ακμή είναι ένα RDD partition και κάθε ακμή είναι ένας μετασχηματισμός πάνω στα δεδομένα. Το γράφημα αυτό λέγεται

άκυκλο επειδή κάποιος μετασχηματισμός δεν μπορεί να επιστρέψει σε ένα παλαιότερο partition.

Το περιβάλλον ανάπτυξης **Apache Spark** χρησιμοποιεί μια master / worker αρχιτεκτονική. Υπάρχει κάποιος **κόμβος** στην συστάδα υπολογιστών ο οποίος παίζει τον ρόλο του «οδηγού» (**driver**) ο οποίος μιλάει με έναν μοναδικό συντονιστή ο οποίος λέγεται «αφέντης» **(master cluster manager)** και διαχειρίζεται τους υπόλοιπους κόμβους της συστάδας υπολογιστών οι οποίοι λέγονται «εργάτες»(**workers**) και στους οποίους τρέχουν οι εκτελεστές εργασίας (**executors**). Αυτή η αρχιτεκτονική φαίνεται στην **εικόνα 35**.



Εικόνα 35:
Master / worker αρχιτεκτονική του **Apache Spark**.

Ο «οδηγός» και οι «εκτελεστές εργασίας» τρέχουν τις δικές τους Java διεργασίες. Μπορούν να τρέχουν στο ίδιο μηχάνημα (οριζόντια συστάδα υπολογιστών) ή σε ξεχωριστά μηχανήματα (κάθετη συστάδα υπολογιστών).

Οι εφαρμογές **Spark** τρέχουν σαν ανεξάρτητα σύνολα από διαδικασίες πάνω σε μια συστάδα υπολογιστών, το οποίο ενορχηστρώνεται από ένα **SparkContext** αντικείμενο το οποίο βρίσκεται στο main πρόγραμμα και ονομάζεται «πρόγραμμα οδηγού»(**driver program**). Συγκεκριμένα για να τρέξει σε μια συστάδα υπολογιστών, το **SparkContext** συνδέεται σε κάποιον cluster manager, είτε στον standalone cluster manager του ίδιου του Spark, είτε στον **Mesos** ή στον **YARN**, από τους οποίους εντοπίζει τις πηγές που θα χρειαστεί για την εφαρμογή που θα τρέξει. Μόλις συνδεδεί, το **Spark** αποκτά «εκτελεστές εργασίας» πάνω στους διαθέσιμους κόμβους, οι οποίοι είναι διεργασίες που εκτελούν υπολογισμούς και αποθηκεύουν δεδομένα για την εφαρμογή που τρέχει στην συστάδα υπολογιστών. Έπειτα, στέλνει τον κώδικα της εφαρμογής (που ορίζεται σαν JAR και περνάει

στο **SparkContext**) στους «εκτελεστές εργασίας». Τέλος το **SparkContext** στέλνει δεδομένα για να εκτελεστούν στους «εκτελεστές εργασίας».

Το πρόγραμμα-οδηγός του **Spark** (**Spark driver**) τρέχει την main μέθοδο της εφαρμογής και είναι το σημείο όπου το **SparkContext** δημιουργείται. Το πρόγραμμα-οδηγός του **Spark** περιέχει διάφορα δομικά στοιχεία όπως ο DAGScheduler, TaskScheduler και BlockManager οι οποίοι είναι υπεύθυνοι για την μετάφραση του **Spark** κώδικα που έχει γράψει ο χρήστης σε πραγματικές εργασίες **Spark** οι οποίες θα εκτελεστούν στην συστάδα υπολογιστών. Το πρόγραμμα-οδηγός του **Spark** είναι υπεύθυνο για τα ακόλουθα

- Τρέχει στο κετρνικό κόμβο της Spark συστάδας, δρομολογεί την εκτέλεση των εργασιών και επικοινωνεί με τον ενορχηστρωτή της συστάδας.
- Μεταφράζει τα RDDs σε γράφημα εκτέλεσης και χωρίζει το γράφημα αυτό σε πολλά στάδια (**stages**).
- Σώζει τα μετά-δεδομένα (metadata) για όλα τα RDDs και τα partitions τους.
- Δημιουργεί τα jobs και τα tasks. Χωρίζει την εφαρμογή του χρήστη σε μικρότερες μονάδες εκτέλεσης που ονομάζονται tasks. Τελικά τα tasks θα εκτελεστούν από τις «μονάδες εργασίας».
- Συλλέγει τα αποτελέσματα από τις «μονάδες εργασίας».

Η «μονάδα εργασίας» είναι ένα κατανεμημένο πρόγραμμα υπεύθυνο για την εκτέλεση των εργασιών. Οι «μονάδες εργασίας» τρέχουν καθ' όλη την διάρκεια μιας **Spark** εφαρμογής αλλά οι χρήστες έχουν την δυνατότητα εάν θελήσουν να προσθέσουν ή να αφαιρέσουν «μονάδες εργασίας» δυναμικά την ώρα που τρέχει η εφαρμογή. Συγκεκριμένα οι «μονάδες εργασίας» είναι υπεύθυνες για

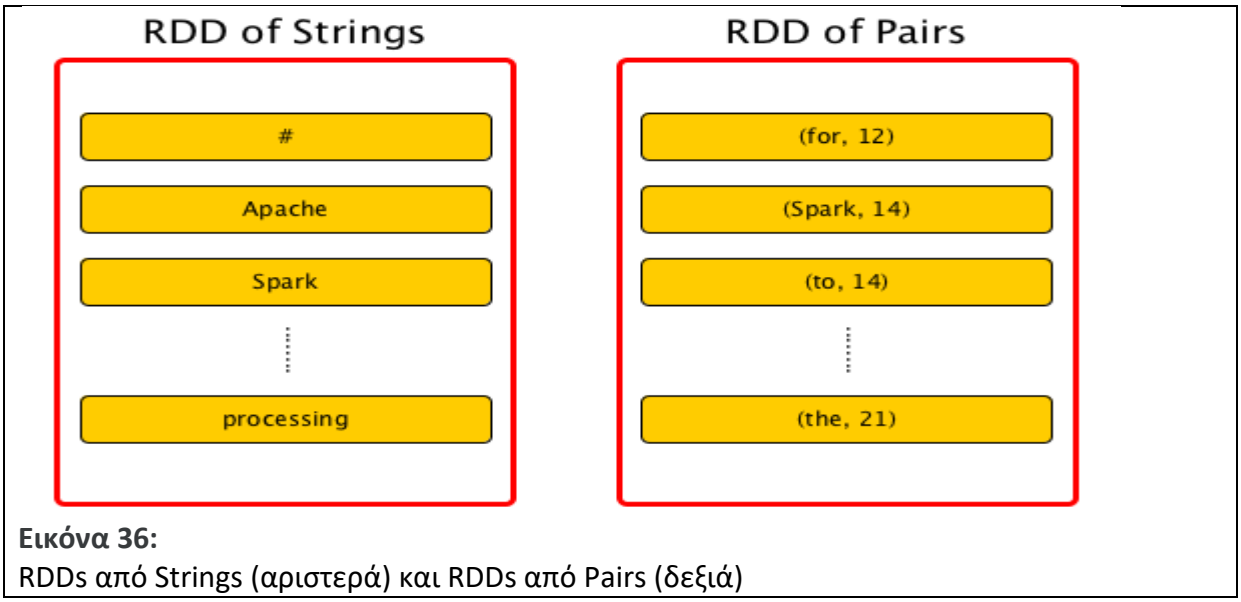
- Την επεξεργασία των δεδομένων (**data processing**)
- Το διάβασμα και την εγγραφή σε εξωτερικές πηγές.
- Την αποθήκευση αποτελεσμάτων από την επεξεργασία των δεδομένων στην μνήμη, στην cache ή στον σκλήρο δίσκο.

Ο διαχειριστής της συστάδας υπολογιστών σε μια **Spark** εφαρμογή είναι μια εξωτερική υπηρεσία η οποία είναι υπεύθυνη για την παροχή των απαραίτητων πόρων (**resources**) σε μια **Spark** συστάδα και την διάθεση τους σε μια **Spark** εργασία. Υπάρχουν τρεις διαφορετικοί τύποι από διαχειριστές συστάδας που είναι διαθέσιμοι σε μια **Spark** εφαρμογή, ο Hadoop YARN, ο Apache Mesos και ο απλός standalone Spark cluster manager. Η επιλογή του κατάλληλου διαχειριστής συστάδας εξαρτάται από τον σκοπό της **Spark** εφαρμογής γιατί κάθε διαχειριστής συστάδας παρέχει διαφορετικά σύνολα από δυνατότητες προγραμματισμού. Ο απλός standalone Spark διαχειριστής συστάδας είναι ο πιο εύκολος στην χρήση σε σχέση με τους υπόλοιπους.

4.5 RDD Δομές δεδομένων

Μία από τις βασικές δομές δεδομένων του Spark είναι τα **Resilient Distributed Datasets** (**RDD**). Είναι μια αμετάβλητη κατανεμημένη δομή δεδομένων από διάφορα αντικείμενα (**objects**). Κάθε σύνολο δεδομένων σε ένα RDD χωρίζεται σε λογικά τμήματα (**partitions**), τα οποία μπορεί να επεξεργάζονται σε διαφορετικά μηχανήματα (**nodes**) σε μια συστάδα υπολογιστών. Τα **RDDs** μπορούν να περιέχουν οποιονδήποτε τύπο δεδομένων της Python, της Java ή της Scala. Τα **RDDs** είναι μια δομή δεδομένων ανεκτική σε σφάλματα (**fault tolerant**) καθώς όταν κάποιο τμήμα καταστραφεί για κάποιο λόγο τότε το σύστημα διαθέτει αρκετή πληροφορία ώστε να το κατασκευάσει ξανά. Το **Spark** χρησιμοποιεί τα **RDDs** για να επιτύχει ταχύτερες και αποδοτικότερες ενέργειες (**operations**) MapReduce.

Στην **Εικόνα 36** φαίνεται παράδειγμα 2 διαφορετικών τύπων **RDDs**.



Εικόνα 36:
RDDs από Strings (αριστερά) και RDDs από Pairs (δεξιά)

Υπάρχουν δύο τρόποι για να δημιουργηθεί ένα **RDD**

- Με τη παραλληλοποίηση μια υπάρχουσας δομής δεδομένων στο πρόγραμμα οδηγό του **Spark**. Στην ουσία το πρόγραμμα οδηγός του **Spark** διαχωρίζει σε τμήματα τα δεδομένα και τα στέλνει στους κόμβους της συστάδας υπολογιστών.
- Από κάποια αρχεία που βρίσκονται σε κάποιο εξωτερικό σύστημα αποθήκευσης όπως ένα κοινόχρηστο σύστημα αρχείων (**shared file system**), ένα HDFS ή οποιαδήποτε πηγή δεδομένων που προσφέρει μια μορφή Hadoop.

Συνοψίζοντας τα **RDDs** έχουν τα ακόλουθα χαρακτηριστικά

- **Αποθήκευση στην κύρια μνήμη (In-memory)**: Τα δεδομένα μέσα σε ένα **RDD** αποθηκεύονται στην μνήμη για όσο περισσότερο χρόνο γίνεται.

- **Αμετάβλητα ή για ανάγνωση μόνο**: Δεν γίνεται να αλλάξουν από την στιγμή που δημιουργηθούν και μπορούν μόνο να μετασχηματιστούν με μετασχηματισμούς σε νέα **RDDs.**
- **Αργή Εκτίμηση (lazy-evaluated)**: Τα δεδομένα μέσα στο **RDD** δεν είναι διαθέσιμα έως ότου εκτελεστεί σε αυτά μια εντολή τύπου «ενέργειας».
- **Αποθηκεύονται στην cache (Cacheable)**: Τα δεδομένα των **RDD** μπορούν να αποθηκεύονται στην μνήμη (που είναι το επιθυμητό για λόγους ταχύτητας) αλλά και στο σκληρό δίσκο.
- **Παράλληλα (Parallel)**: Τα **RDDs** μπορούν να επεξεργάζονται παράλληλα.
- **Συγκεκριμένου τύπου (Typed):** Τα **RDDs** είναι συγκεκριμένου τύπου δεδομένα για παράδειγμα RDD[Float] ή RDD[(Int, String)]
- **Τμηματικά (Partitioned)**: Τα **RDDs** διαχωρίζονται σε λογικά τμήματα τα partitions και μοιράζονται σε όλους τους κόμβους της συστάδας υπολογιστών.

4.6 Spark SQL, DataFrames and Datasets

Το **Spark SQL** είναι ένα δομικό στοιχείο του **Spark** το οποίο χρησιμοποιείται για επεξεργασία δομημένων δεδομένων. Σε αντίθεση με το API των **RDD**, το API που παρέχει το **Spark SQL** περιέχει περισσότερες πληροφορίες για την δομή τόσο των δεδομένων όσο και των υπολογισμών που θα γίνουν. Εσωτερικά το **Spark SQL** εκμεταλλεύεται αυτές τις πληροφορίες για να βελτιώσει την απόδοση της εφαρμογής. Υπάρχουν πολλοί τρόποι αλληλεπίδρασης με το **Spark SQL** συμπεριλαμβανομένου της **SQL** αλλά και του Dataset API. Μια χρήση του **Spark SQL** είναι η εκτέλεση **SQL queries**, τα οποία όταν εκτελούνται μέσα από κάποια γλώσσα προγραμματισμού πάντα επιστρέφουν τα αποτελέσματα σαν **DataFrames** ή σαν **Datasets**.

Dataset είναι μια κατανεμημένη συλλογή από δεδομένα. Το **Dataset** είναι μια δομή η οποία προστέθηκε στην έκδοση 1.6 του **Spark** και παρέχει όλα τα πλεονεκτήματα των **RDDs** (strong typing, χρήση εκφράσεων lamdas) με τα οφέλη της βελτιωμένης **Spark SQL** μηχανής επεξεργασίας δεδομένων. Ένα **Dataset** μπορεί να κατασκευαστεί από JVM αντικείμενα και να χρησιμοποιηθεί από συναρτησιακούς μετασχηματισμούς (όπως map, flatMap, filter κα). Το **Dataset API** είναι διαθέσιμο σε Scala και Java. Τα **Dataset** εκτιμώνται με καθυστέρηση δηλαδή ο υπολογισμός τους πυροδοτείται μόνο όταν καλεστεί κάποια ενέργεια.

Dataframe είναι ένα **Dataset** το οποίο οργανώνεται σε στήλες με όνομα. Είναι ισοδύναμο με ένα πίνακα σε μια σχεσιακή βάση δεδομένων ή με ένα data frame στην Python και την R, αλλά με πολλές πλούσιες βελτιστοποιήσεις. Τα **Dataframes** μπορούν να κατασκευαστούν από πολλές πηγές δεδομένων όπως δομημένα αρχεία δεδομένων, πίνακες σε μια βάση δεδομένων σαν το Hive, από εξωτερικές βάσεις δεδομένων και

υπάρχοντα **RDDs**. Το **Dataframe API** είναι διαθέσιμο σε Scala, Java, Python και R. Στην Scala και στην Java ένα **Dataframe** παρουσίαζεται σαν **Dataset** από **Row**. Δηλαδή στην Java σαν Dataset<Row> και στην Scala σαν DataSet[Row].

4.7 Βασικές μέθοδοι του Apache Spark

Οι μέθοδοι (**operations**) στο **Spark** (και για τα **RDDs** και τα **Datasets**) χωρίζονται σε μετασχηματισμούς (**transformations**) και ενέργεις (**actions**). Οι μετασχηματισμοί θα παράξουν νέα **RDDs** ή **Datasets** και δεν μπορούν να τροποποιήσουν τα δεδομένα εισόδου, και οι ενέργειες θα πυροδοτήσουν τον υπολογισμό και θα επιστρέψουν τα αποτελέσματα. Οι ενέργειες ακόμα μεταφέρουν τα δεδομένα που έχουν υπολογιστεί από τους κόμβους της συστάδας υπολογιστών, δηλαδή τις μονάδες εργασίας (**executors**), στο πρόγραμμα-οδηγό (**driver**) του **Spark**. Αυτές οι μέθοδοι (**operations**) ταιριάζουν με το MapReduce του Hadoop, καθώς το Map υλοποείται από τα μετασχηματισμούς του **Spark** και το Reduce από τις ενέργειες**.**

Ακόμα υπάρχουν συγκεκριμένες μέθοδοι στο **Spark** που πυροδοτούν το "ανακάτεμα" (**shuffling**) των δεδομένων. Το "ανακάτεμα" είναι ένας μηχανισμός του **Spark** για να ξανά-κατανήμει (**re-distribute**) τα δεδομένα που έχουν ομαδοποιηθεί διαφορετικά σε όλα τα τμήματα. Αυτό συνήθως περιλαμβάνει αντιγραφή των αρχείων ανάμεσα στις μονάδες εργασίας (**executors**) και στα μηχανήματα της συστάδας υπολογιστών, κάτι το οποίο αποτελεί και μια πολύ ακριβή διαδικασία.

Ορισμένα παραδείγματα μετασχηματισμών **Spark** είναι:

- **map**
- **filter**
- **flatMap**
- **mapPartitions**
- **mapPartitionsWithIndex**
- **sample**
- **union**
- **intersection**
- **distinct**
- **groupByKey**
- **reduceByKey**
- **aggregateByKey**
- **sortByKey**
- **join**
- **cartesian**

Ορισμένα παραδείγματα ενεργειών του **Spark** είναι:

- **reduce**

- **collect**
- **count**
- **first**
- **take(n)**
- **takeSample**
- **saveAsTextFile**
- **countByKey**
- **foreach**

4.8 Κοινές μεταβλητές και αποθήκευση μεταβλητών στο Spark

Όπως φαίνεται στα προηγούμενα κεφάλαια, μια μεγάλη αφαίρεση (**abstraction**) που προσφέρει το **Apache Spark** είναι τα **Resilient Distributed Datasets** (**RDD**), τα οποία είναι μια συλλογή δεδομένων που κατανέμονται σε όλα τους κόμβους (**nodes**) της συστάδας υπολογιστών και μπορούν να επεξεργαστούν παράλληλα. Ο προγραμματιστής έχει την δυνατότητα να αποθηκεύσει τα **RDDs** στην μνήμη κάθε κόμβου ώστε να χρησιμοποιείται αποδοτικότερα στην παράλληλη επεξεργασία των δεδομένων.

Μια δεύτερη αφαίρεση που προσφέρει το **Apache Spark** είναι οι κοινές μεταβλητές (**shared variables**) που μπορούν να χρησιμοποιήσουν παράλληλες ενέργειες πάνω στα δεδομένα. Όταν το **Spark** τρέχει κάποια συνάρτηση παράλληλα σαν ένα σύνολο από ενέργειες (**tasks**) σε διαφορετικούς κόμβους της συστάδας υπολογιστών, στέλνει ένα αντίγραφο κάθε μεταβλητής που χρησιμοποείται στην συνάρτηση σε κάθε ενέργεια. Κάποιες φορές, αυτή η μεταβλητή χρειάζεται να είναι κοινή ανάμεσα σε όλες τις ενέργειες ή και σε όλους τους κόμβους. Το **Spark** διαθέτει δύο είδη κοινών μεταβλητών.

- τις **broadcast variables**, που μπορούν να χρησιμοποιηθούν για να αποθηκευτεί μια τιμή μόνο για ανάγνωση της μεταβλητής στην μνήμη όλων των κόμβων της συστάδας υπολογιστών, παρά να μεταφέρεται ένα αντίγραφο στο κόμβο που θα εκτελέσει την ενέργεια. Για παράδειγμα μπορεί να δωθεί σε ένα κόμβο ένα πολύ μεγάλο αντίγραφο ενός συνόλου δεδομένων αποφεύγοντας έτσι το πολύ μεγάλο φόρτο στο δίκτυο της συστάδας υπολογιστών. Το **Spark** διαθέτει διάφορους αλγορίθμους για μειώνει το κόστος τις επικοινωνίας μταξύ των κόμβων και του κετρικού προγράμματος οδηγού του **Spark**. Στον παρακάτω Scala κώδικα φαίνεται η δημιουργία και η χρήση μιας **broadcast variable.**

```
val spark = SparkSession.builder().getOrCreate()
val broadcastVar = spark.sparkContext.broadcast(Array(1, 2, 3))
broadcastVar.value
```

- τους συσσωρευτές (**accumulators**), που είναι μεταβλητές στις οποίες μπορούν να προστεθούν τιμές, όπως για παράδειγμα μετρητές και αθροίσματα. Στον παρακάτω Scala κώδικα φαίνεται η δημιουργία και η χρήση ενός συσσωρευτή**.**

```
val spark = SparkSession.builder().getOrCreate()
val accum = spark.sparkContext.accumulator(0, "Accumulator Example")
spark.sparkContext.parallelize(Array(1, 2, 3)).foreach(x => accum += x)
accum.value
```

Εκτός από τις **broadcast** μεταβλητές και τους συσσωρευτές, ο χρήστης του **Spark** έχει την δυνατότητα να αποθηκεύσει **RDD** στην μνήμη. Αυτή η τεχνική ονομάζεται **caching** ή **persistence** και είναι τεχνικές βελτιστοποίησης για επαναλαμβανόμενους **Spark** υπολογισμούς. Συγκεκριμένα σώζουν ενδιάμεσα αποτελέσματα ώστε να μπορούν να επαναχρησιμοποιηθούν σε επόμενα στάδια. Τα αποτελέσματα αυτά μπορούν να σωθούν στην μνήμη ή σε κάποιον σκληρό δίσκο.

Τα **RDDs** μπορούν να αποθηκευτούν χρησιμοποιώντας τις εντολές **cache()** ή **persist()**. Η διαφορά μεταξύ αυτών των δύο εντολών είναι μόνο συντακτική. Η **cache()** είναι ακριβώς ίδια με την εντολή persist(MEMORY_ONLY). Ακόμα τα **RDDs** μπορούν να διαγραφούν από την μνήμη ή τον σκληρό δίσκο που αποθηκεύτηκαν χρησιμοποιώντας την εντολή **unpersist()**.

5. Υλοποίηση του αλγόριθμου Alpha

Στο κεφάλαιο αυτό γίνεται μια παρουσίαση του κώδικα Scala που υλοποιήθηκε ο αλγόριθμος Alpha αλλά και ο τρόπος που επιβεβαιώνεται η ορθή λειτουργία του αλγορίθμου.

5.1 Κώδικας Υλοποίησης

Για την υλοποίηση του αλγόριθμου **Alpha** χρησιμοποιήθηκε η γλώσσα προγραμματισμού **Scala**, η οποία παρουσιάστηκε σε προηγούμενο κεφάλαιο. Υλοποιήθηκαν δύο εκδόσεις του αλγόριθμου **Alpha**, η μία χρησιμοποιώντας το **Περιβάλλον Ανάπτυξης Εφαρμογών Spark** και η άλλη χωρίς την χρήση του εν λόγω περιβάλλοντος. Σκοπός είναι η μέτρηση των χρόνων και για τις δύο υλοποιήσεις, ώστε να γίνει εξαγωγή συμπερασμάτων για το πότε είναι χρήσιμη η χρήση του **Spark** και πότε όχι.

Σαν είσοδο δεδομένων (**input dataset**) χρησιμοποιήθηκε ένα σύνολο γεγονότων από έναν οργανισμό τηλεπικοινωνιών. Το συγκεκριμένο csv αρχείο έχει μέγεθος περίπου 400mb και περιέχει 36100 ίχνη δραστηριοτήτων (traces).

Πολλά από τα παραπάνω ίχνη δραστηριοτήτων επαναλαμβάνονται πάρα πολλές φορές για αυτό υλοποιήθηκε κώδικας ο οποίος είναι παραμετροποιήσιμος και μπορεί να τροφοδοτήσει στον αλγόριθμο **Alpha** τα πιο συχνά εμφανιζόμενα ίχνη δραστηριοτήτων ή έναν συγκεκριμένο αριθμό από ίχνη (ο οποίος προσδιορίζεται από το χρήστη) ή αλλιώς όλα τα ίχνη που διαθέτει το σύνολο γεγονότων. Αυτή η υλοποίηση είναι απαραίτητη γιατί το παραγόμενο αποτέλεσμα του αλγόριθμου **Alpha** δεν εξαρτάται από το πόσες φορές εμφανίζεται ένα ίχνος στο σύνολο γεγονότων**.** Ακόμα κάποια πολύ σπάνια εμφανιζόμενα ίχνη μπορεί να μην είναι επιθυμητό να επεξεργαστούν από τον αλγόριθμο **Alpha** καθώς ενδέχεται να αλλάξουν το τελικό δίκτυο Petri κατά πολύ και τελικά να μην απεικονίζει τις συνήθεις διαδικασίες (**processes**).

```
//Κώδικας 1
def main(args: Array[String]): Unit = {
  Logger.getLogger("org").setLevel(Level.ERROR)
  val traceTools: TraceTools = new TraceTools()
  val logPath = "src/main/resources/data.csv"
  val numOfTraces = 3
  val percentage : Float = 1
  val readAll : Boolean = false
  val filtering : Boolean = true

  val spark = SparkSession
    .builder()
    .appName("AlphaAlgorithm")
    .master("local[*]")
    .getOrCreate()
```

```
  val tracesDS : Dataset[(String, List[String])] =
traceTools.getTracesToInspect(logPath, numOfTraces, readAll, filtering, percentage)

  val petriNet: PetriNet = executeAlphaAlgorithm(tracesDS)
  println(petriNet)

  // Stop the session
  spark.stop()
}
```

Στον παραπάνω κώδικα (**Κώδικας 1**) φαίνεται η **main** συνάρτηση από την οποία ξεκινάει ο αλγόριθμος **Alpha** καθώς και οι διάφοροι παράμετροι που πρέπει να επιλέξει ο χρήστης ώστε να ξεκινήσει η εκτέλεση του αλγόριθμου **Alpha**. Αυτές οι μεταβλητές είναι

- **logPath:** Το path που θα διαβάσει το πρόγραμμα το σύνολο γεγονότων που θα επεξεργαστεί.
- **readAll:** Boolean μεταβλητή η οποία αν είναι true διαβάζει όλα τα ίχνη δραστηριοτήτων από το σύνολο γεγονότων, αλλιώς θα διαβάσει τόσα ίχνη όσα έχουν δηλωθεί στην μεταβλητή **numOfTraces.**
- **filtering:** Boolean μεταβλητή η οποία αν είναι true φιλτράρει τα ίχνη δραστηριοτήτων και κρατάει μόνο όσα έχουν ποσοστό εμφάνισης άνω της τιμής της μεταβλητής **percentage.**
- **percentage:** Ποσοστό που δείχνει ποιο είναι το κάτω όριο ποσοστού εμφάνισης ώστε να επεξεργαστεί ένα ίχνος από τον αλγόριθμο **Alpha.** Αν για παράδειγμα η τιμής της μεταβλητής είναι 1, τότε αν και το **filtering** είναι **true,** ο αλγόριθμος **Alpha** θα επεξεργαστεί μόνο τα ίχνη δραστηριοτήτων που έχουν ποσοστό εμφάνισης πάνω από 1%.
- **numOfTraces:** Ο αριθμός των ιχνών δραστηριοτήτων που θα επεξεργαστεί ο αλγόριθμος **Alpha**. Για να λάβει το πρόγραμμα υπόψιν αυτήν την παράμετρο πρέπει το **readAll** να είναι **false**.

Ακόμα στην main μέθοδο αρχικοποιείται το **SparkSession**, τα ίχνη δραστηριοτήτων μέσω της μεθόδου **getTracesToInspect,** πυροδοτείται ο αλγόριθμος **Alpha** μέσω της μεθόδου **executeAlphaAlgorithm** και εκτυπώνεται το παραγόμενο δίκτυο Petri.

```
//Κώδικας 2
def readAllTracesFromCsvFile(path: String) : Dataset[(String, List[String])] = {
  val spark = SparkSession.builder().getOrCreate()
  import spark.implicits._

  val df = spark.read.format("csv").option("header", "true").load(path)

  //Dataset[(String, List[String])]
  return df.select("orderid", "eventname", "starttime", "endtime", "status")
    .filter(df("status").isin(List("Completed"):_*))
    .orderBy("starttime")
    .map(x=>(x.get(0).toString,x.get(1).toString))
    .groupByKey(x=>x._1)
    .mapGroups{case(k, iter) => (k, iter.map(x => x._2).toList)}
}
```

Στον **Κώδικα 2** παρουσιάζεται η μέθοδος η **readAllTracesFromCsvFile** οποία διαβάζει όλα τα ίχνη δραστηριοτήτων από το σύνολο γεγονότων (ένα csv file εν προκειμένω). Συγκεκριμένα διαλέγει κάποιες στήλες από το csv file, φιλτράρει τις γραμμές ώστε να κρατήσει μόνο όσες έχουν status "**Completed**" (καθώς μόνο τα συγκεκριμένα events πρέπει να επεξεργαστούν), τις τοποθετεί σε σειρά σύμφωνα με το πότε ξεκίνησαν "**starttime**" και τέλος τις ομαδοποιεί σύμφωνα με το **orderid** (με τους συναρτησιακούς τελεστές **groupByKey** και **mapGroups**). Έτσι δημιουργεί ίχνη δραστηριοτήτων με την μορφή **Dataset[(String, List[String])]**, όπου το πρώτο στοιχείο του tuple είναι το **orderid** και το δεύτερο είναι μια λίστα από δραστηριότητες.

```
//Κώδικας 3
def readSpecificNumberOfTracesFromCsvFile(path: String, numOfTraces: Int) :
Dataset[(String, List[String])] = {
  val spark = SparkSession.builder().getOrCreate()
  import spark.implicits._

  val df = spark.read.format("csv").option("header", "true").load(path)

  val orderIds = df.select("orderid")
    .distinct()
    .limit(numOfTraces)
    .as(Encoders.STRING)
    .collect()
    .toList

  //Dataset[(String, List[String])]
  return df.select("orderid", "eventname", "starttime", "endtime", "status")
    .where( df("orderid").isin(orderIds:_*))
    .filter(df("status").isin(List("Completed"):_*))
    .orderBy("starttime")
    .map(x=>(x.get(0).toString,x.get(1).toString))
    .groupByKey(x=>x._1)
    .mapGroups{case(k, iter) => (k, iter.map(x => x._2).toList)}
}
```

Ομοίως ο **Κώδικας 3**, διαβάζει έναν συγκεκριμένο αιρθμό από ίχνη δραστηριοτήτων. Ο αριθμός αυτός δίνεται σαν είσοδος στην μέθοδο (**numOfTraces**) και επιστρέφει τα ίχνη στην μορφή **Dataset[(String, List[String])]**. Η διαφορά με τον **Κώδικα 2** είναι ότι βρίσκει από την αρχή τα orderIds που θα επιστρέψει σαν ίχνη και έπειτα εκτελεί ένα **where operation** ώστε να κρατήσει μόνο τα ίχνη με τα συγκεκριμένα orderIds.

Έχοντας υπολογίσει πλέον τα ίχνη δραστηριοτήτων (έστω tracesDS), το επόμενο βήμα του προγράμματος είναι να φιλτράρει τα ίχνη ανάλογα με την boolean παράμετρο **filtering**. Αν η **filtering** έχει τιμή true τότε τα αρχικό σύνολο από ίχνη δραστηριοτήτων θα επεξεργαστεί ώστε να διαθέτει μόνο τα ίχνη εκείνα που ικανοποιούν την παράμετρο **percentage**. Η διαδικασία αυτή του φιλτραρίσματος του αρχικού συνόλου από ίχνη δραστηριοτήτων φαίνεται στον **Κώδικας 4.**

```
//Κώδικας 4
def filterTraces(tracesDS: Dataset[(String, List[String])], percentage: Float):
Dataset[(String, List[String])] = {
  implicit def listStringEncoder: org.apache.spark.sql.Encoder[List[String]] =
org.apache.spark.sql.Encoders.kryo[List[String]]
  implicit def tupleListStringEncoder: org.apache.spark.sql.Encoder[(String, List[String])] =
org.apache.spark.sql.Encoders.kryo[(String, List[String])]
  implicit def longStringEncoder: org.apache.spark.sql.Encoder[(List[String], Long)] =
org.apache.spark.sql.Encoders.kryo[(List[String], Long)]
  implicit def floatStringEncoder: org.apache.spark.sql.Encoder[(List[String], Float)] =
org.apache.spark.sql.Encoders.kryo[(List[String], Float)]

  val initNumberOfTraces = tracesDS.count()
  println("Initial number of traces = " + initNumberOfTraces)

  import org.apache.spark.sql.functions._
  val tracesToInspect = tracesDS
    .toDF("traceId", "trace")
    .groupBy("trace")
    .agg(count("trace"))
    .map(trace=>(trace.getAs[Seq[String]]("trace").toList,trace.getAs[Long]("count(trace)")))
    .filter(trace => (trace. 2.toFloat / initNumberOfTraces) > (percentage/100) )
    .map(trace=>("xxx", trace._1))

  println("Number of traces to inspect = " + tracesToInspect.count())
  tracesToInspect
}
```

Στην αρχή του **Κώδικα 4** τέσσερεις **encoders** οι οποίοι είναι απαραίτητοι για την εκτέλεση του προγράμματος **Spark**. Ο **encoder** είναι ένα βασικό στοιχείο για το serialization και το deserialization των δεδομένων στο **Spark** ώστε να είναι δυνατή η αποστολή τους από το κεντρικό πρόγραμμα οδηγό του Spark στους κόμβους της συστάδας σε όσο το δυνατόν μικρότερο χρονικό διάστημα αλλά και σε όσο το δυνατόν μικρότερο μέγεθος. Στον **Κώδικα 4** έχουν δημιουργηθεί **encoders** για τους παρακάτω τύπους Scala δεδομένων, οι οποίοι επεξεργάζονται ώστε να κατασκευαστεί το **tracesToInspect (**τύπου **Dataset[(String, List[String])]).** Οι τύποι δεδομένων είναι

- **List[String] :** λίστα από String.
- **(String , List[String]) :** Tuple που περίεχει String στο πρώτο στοιχείο και λίστα από String στο δεύτερο στοιχείο.
- **(List[String] , Long) :** Tuple που περίεχει λίστα από String στο πρώτο στοιχείο και Long στο δεύτερο στοιχείο.
- **(List[String] , Float) :** Tuple που περίεχει λίστα από String στο πρώτο στοιχείο και Float στο δεύτερο στοιχείο.

Για τον υπολογισμό του φιλτραρισμένου συνόλου γεγονότων, όπως φαίνεται στον **Κώδικα 4,** ακολουθείται η παρακάτων διαδικασία

- Μετατροπή του αρχικό Dataset από ίχνη δραστηριοτήτων σε Dataframe με στήλες **traceId** και **trace** (**toDF** operation)
- Ομαδοποίηση των γραμμών του Dataframe σύμφωνα με την δεύτερη στήλη **trace** (**groupBy** operation). Έτσι ο κώδικας βρίσκει μοναδικά ίχνη και υπολογίζει πόσες φορές εμφανίζετε το καθένα (**agg** operation).

- Μετασχηματισμός των δεδομένων σε ένα Tuple με πρώτο στοιχείο το ίχνος και δεύτερο στοιχείο των αριθμό των εμφανίσεων του στο σύνολο γεγονότων (πρώτο **map** operation).
- Φιλτράρισμα των δεδομένων ανάλογα με το ποσοστό εμφάνισης στο σύνολο γεγονότων. Αν το ποσοστό εμφάνισης (**trace._2.toFloat / initNumberOfTraces**) είναι πάνω από την τιμή **percentage** τότε το ίχνος θα επεξεργαστεί από τον **Alpha Algorithm** (τελεστής **filter**).

Μέχρι αυτό το σημείο έχει γίνει ο υπολογισμός των ιχνών δραστηριοτήτων που θα επεξεργαστούν από τον αλγόριθμο Alpha. Τα βήματα εκτέλεσης του αλγορίθμου φαίνονται στον **Κώδικα 5.** Στον κώδικα φαίνονται τα 8 βήματα που ακολουθεί ο αλγόριθμος Alpha ώστε να υπολογίσει το τελικό δίκτυο Petri από το σύνολο γεγονότων που δέχεται σαν είσοδο (**traceDS**). Συγκεκριμένα κάθε βήμα του αλγορίθμου αντιστοιχεί και στην κλήση μιας συνάρτησης, εκτός από το βήμα 4 που λόγω της πολυπλοκότητας του έχει σπάσει σε δύο διαφορετικές μεθόδους, μία για τον υπολογισμό του πίνακα σχέσεων (**footprint graph**) και η δεύτερη για τον υπολογισμό των **causal groups**.

```
//Κώδικας 5
def executeAlphaAlgorithm(tracesDS : Dataset[(String, List[String])]) : PetriNet = {

  val steps : AlphaAlgorithmSteps = new AlphaAlgorithmSteps()
  tracesDS.cache()

  //Step 1 - Find all transitions / events, Sorted list of all event types
  val events = steps.getAllEvents(tracesDS)

  //Step 2 - Construct a set with all start activities (Ti)
  val startActivities = steps.getStartActivities(tracesDS)

  //Step 3 - Construct a set with all final activities (To)
  val finalActivities = steps.getFinalActivities(tracesDS)

  //Step 4 - Footprint graph - Causal groups
  val logRelations : Dataset[(Pair, String)] = steps.getFootprintGraph(tracesDS, events)
  tracesDS.unpersist()
  val causalGroups : Dataset[CausalGroup[String]] = steps.getCausalGroups(logRelations)

  //Step 5 - compute only maximal groups
  val maximalGroups : List[CausalGroup[String]] = steps.getMaximalGroups(causalGroups)

  //step 6 - set of places/states
  val places : Places = steps.getPlaces(maximalGroups, startActivities, finalActivities)

  //step 7 - set of arcs (flow)
  val edges : List[Edge] = steps.getEdges(places)

  //step 8 - construct petri net
  return new PetriNet(places, events, edges)
}
```

Το πρώτο βήμα του αλγορίθμου είναι να βρεθούν όλες οι δυνατές δραστηριότητες στο σύνολο γεγονότων που πρέπει να επεξεργαστεί. Αυτό συμβαίνει στον **Κώδικα 6**. Συγκεκριμένα από όλα τα ίχνη δραστηριοτήτων στο Dataset tracesDS (με την χρήση του τελεστή **map**) ξεχωρίζονται οι δραστηριότητες οι οποίες αποτελούν το ίχνος (τελεστής **flatMap**) και και στην συνέχεια μοναδικοποιούνται (toSet μέθοδος).

```
//Κώδικας 6
def getAllEvents(tracesDS: Dataset[(String, List[String])]) : List[String] = {
  return tracesDS
    .map(x=>x._2)
    .flatMap(x=>x.toSet)
    .collect()
    .toSet
    .toList
    .sorted
}
```

Το δεύτερο βήμα του αλγορίθμου είναι να κατασκευαστεί ένα σύνολο από όλες τις αρχικές δραστηριότητες δηλαδή ένα σύνολο από τις αρχικές δραστηριότητες όλων των ιχνών. Αυτό συμβαίνει στον **κώδικα 7**.

```
//Κώδικας 7
def getStartActivities(tracesDS: Dataset[(String, List[String])]) : Set[String] = {
  return tracesDS.map(x=>x._2.head).collect().toSet
}
```

Το τρίτο βήμα του αλγορίθμου είναι να κατασκευαστεί ένα σύνολο από όλες τις τελικές δραστηριότητες δηλαδή ένα σύνολο από τις τελικές δραστηριότητες όλων των ιχνών. Αυτό συμβαίνει στον **κώδικα 8**.

```
//Κώδικας 8
def getFinalActivities(tracesDS: Dataset[(String, List[String])]) : Set[String] = {
  return tracesDS.map(x=>x._2.last).collect().toSet
}
```

Στην συνέχεια ο αλγόριθμος Alpha πρέπει να κατασκευάσει τον πίνακα σχέσεων ο οποίος περίεχει τις σχέσεις μεταξύ των δραστηριοτήτων. Αυτό συμβαίνει στον **κώδικα 9**.

```
//Κώδικας 9
/**
  * Step 4 Calculate pairs - Footprint graph
  * construct a list of pair events for which computations must be made
  * @param tracesDS
  * @param events
  * @return
  */
def getFootprintGraph(tracesDS: Dataset[(String, List[String])], events: List[String]):
Dataset[(Pair, String)] = {
  val followRelation: FindFollowRelation = new FindFollowRelation()
  val findLogRelations: FindLogRelations = new FindLogRelations()
  val traceTools: TraceTools = new TraceTools()
  val pairsToExamine = traceTools.constructPairsForComputationFromEvents(events)

  /**
    * pairInfo is in the following form
    * AB,PairNotation(DIRECT, FOLLOW)
    * AB,PairNotation(INVERSE, FOLLOW)
    */
  val pairInfo = tracesDS
    .map(traces => followRelation.findFollowRelation(traces, pairsToExamine))
    .map(x=>x.getPairsMap())
    .flatMap(map=>map.toSeq)  //map to collection of tuples
    .map(x=> List(new PairInfo((x._1, new PairNotation(x._2._1.pairNotation))), new
PairInfo((x._1, new PairNotation(x._2._2.pairNotation)))))
    .flatMap(x=>x.toSeq)

  pairInfo.cache()

  /**
```

```
    * relations in  the following form, Footprint graph
    * (FB,CAUSALITY)
    * (BB,NEVER FOLLOW)
    * (AB,PARALLELISM)
    */
  val logRelations = pairInfo
    .groupByKey(x=> x.getPairName())
    .mapGroups{case(k, iter) => (new Pair(k.getFirstMember().toString,
k.getSecondMember().toString), iter.map(x => x.getPairNotation()).toSet)}
    .map(x=>findLogRelations.getDirectAndInverseFollowRelations(x))
    .map(x=>findLogRelations.extractFootPrintGraph(x._1, x._2, x._3))

  pairInfo.unpersist()
  return logRelations
}
```

Στον **κώδικα 9** φαίνεται ότι το πρώτο βήμα για την κατασκευή του πίνακα σχέσεων (footprint graph) είναι η εύρεση των σχέσεων απευθείας διαδοχής (**direct succession**) μεταξύ των δραστηριοτήτων των ιχνών. Οι σχέσεις αυτές στον κώδικα είναι η μεταβλητή **pairInfo** η οποία είναι τύπου **Dataset[PairInfo]**. Το **PairInfo** είναι μια κλάση η οποία περιέχει ένα tuple στο οποίο το πρώτο στοιχείο είναι ένα ζεύγος από δραστηριότητες (κλάση **Pair**) και το δεύτερο στοιχείο είναι μια άλλη κλάση η οποία δείχνει αν το ζεύγος από δραστηριότητες είναι ευθή ή αντίστροφο καθώς και το είδος της σχέσης (κλάση **PairNotation**). Για παράδειγμα, έστω ότι η μεταβλητή **pairInfo** περιέχει τα ακόλουθα δεδομένα σε μορφή Tuples

(AB, PairNotation(DIRECT, FOLLOW))
(AB, PairNotation(INVERSE, FOLLOW))

Από την παραπάνω αναπαράσταση για το ζεύγος δραστηριοτήτων AB, γίνεται αντιληπτό ότι ισχύει A>B και B>A, καθώς για το AB το αντίστοιχο PairNotation με Directionality DIRECT έχει σχέση FOLLOW. Το ίδιο ακριβώς ισχύει και για δεύτερο Tuple που αναφέρεται στο Directionality INVERSE.

Για τον υπολογισμό της μεταβλητής **pairInfo** εκτελείται η διαδικάσια που φαίνεται στον κώδικα 9 και συγκεκριμένα πρώτα υπολογίζονται τα δυνατά ζεύγη που πρέπει να εξεταστούν. Ο υπολογισμός αυτών των ζευγών φαίνεται στο **κώδικα 10.** Συγκεκριμένα για όλες τις πιθανές δραστηριότητες που υπάρχουν στο σύνολο γεγονότων υπολογίζουμε όλα τα δυνατά ζεύγη τα οποία δεν μπορεί να έχουν ίδια τα μέλη του ζεύγους και το αριστερό μέλος να μην είναι μεγαλύτερο από το δεξί μέλος του ζεύγους (συνθήκη **idxX == idxY || idxX < idxY**).

Έτσι για ένα πιθανό σύνολο από δραστηριότητες (A,B,C,D,E) τα εξεταζόμενα ζεύγη θα είναι τα (AA, AB, AC, AD, BB, BC, BD, CC, CD, DD).

```
//Κώδικας 10
/**
  * We assume that the events list contains no duplicates and they are sorted
  * If the events are A,B,C,D,E then pairs for computation are
  * AA, AB AC AD
  * BB BC BD
  * CC CD
  * DD
```

```
   * @param events
   * @return
   */
def constructPairsForComputationFromEvents(events: List[String]): List[Pair] = {
  for {
    (x, idxX) <- events.zipWithIndex
    (y, idxY) <- events.zipWithIndex
    if (idxX == idxY || idxX < idxY)
  } yield new Pair(x,y)
}
```

Στην συνέχεια τα ζεύγη αυτά τροφοδοτούνται σαν είσοδο στην κλάση FindFollowRelation η οποία για κάποιο ίχνη υπολογίζει τις σχέσεις απευθείας διαδοχής (**direct succession / follow relation**) μεταξύ των δραστηριοτήτων του ίχνους. Αυτό φαίνεται στον **κώδικα 11**, όπου κατασκευάζεται ένα Map με όλα τα ζεύγη που υπολογίστηκαν στο προηγούμενο βήμα. Αυτό το Map περίεχει σαν key το ζεύγος από δραστηριότητες και σαν value ένα Tuple της μορφής (PairNotation, PairNotation) όπου το πρώτο στοιχείο του Tuple περιέχει πληρφορία για την σχέση του DIRECT ζεύγους και το δεύτερο στοιχείο για την σχέση του INVERSE ζεύγους. Η μέθοδος **findFollowRelation** δέχεται σαν όρισμα μια παράμετρο **traceWithCaseId** η οποία είναι της μορφής (String, List[String]). Για παράδειγμα μια τιμή για την παράμετρο αυτή θα μπορούσε να ήταν η (case1, ABCD). Η μέθοδος αυτή έχει σκοπό να βρει τις σχέσεις ακολουθίας (**FOLLOW**) και μη- ακολουθίας **(NOT_FOLLOW)** μεταξύ των δραστηριοτήτων αυτού του ίχνους (δηλαδή των A B C D) και να τις μετατρέψει στην μορφή (PairAB, (DIRECT, FOLLOW/NOT_FOLLOW), (INVERSE, FOLLOW/NOT_FOLLOW)).

```
//Κώδικας 11
def findFollowRelation(traceWithCaseId: (String, List[String]), pairsToExamine:
List[Pair]):FullPairsInfoMap = {
  var pairInfoMap = pairInfoInit(pairsToExamine)

  Range(0, traceWithCaseId. 2.length-1)
    .map(i=> new Pair(traceWithCaseId._2(i), traceWithCaseId._2(i+1)))
    .map(pair=> {
        val pairExists : Boolean = checkIfPairExists(pair, pairInfoMap)
        pairInfoMap = matchPairExists(pairExists, pair, pairInfoMap)
      }
    )

  return new FullPairsInfoMap(pairInfoMap)
}
```

Με τον τρόπο που αναλύθηκε παραπάνω η μεταβλητή **pairInfo** πλέον διαθέτει όλες τις πληροφορίες για τις σχέσεις ακολουθίας (**FOLLOW**) και μη- ακολουθίας **(NOT_FOLLOW)** για όλα τα ζεύγη δραστηριοτήτων που μπορούν να προκύψουν για το σύνολο γεγονότων που έχει δωθεί σαν είσοδο στον αλγόριθμο **Alpha**. Η μεταβλητή **pairInfo** επειδή θα χρησιμοποιηθεί και σε επόμενο στάδιο για τον υπολογισμό του πίνακα σχέσεων (footprint graph), αποθηκεύεται στην μνήμη κάθε κόμβου της συστάδας υπολογιστών με την εντολή **pairInfo.cache()** και όταν πλέον δεν θα είναι απαραίτητη θα διαγραφεί από την μνήμη με την εντολή **pairInfo.unpersist()**.

Ο υπολογισμός του πίνακα σχέσεων γίνεται στην μεταβλητή **logRelations** (βρίσκεται στον **κώδικα 9**). Σκοπός είναι για τα διαθέσιμα ζεύγη από την μεταβλητή **pairInfo** να βρεθούν οι υπόλοιπες σχέσεις που πρέπει να υπάρχουν στο πίνακα όπως οι «αιτιατότητα» **(Causality x->y)**, η παραλληλία **(Parallel x||y)** και η επιλογή (**Choice x#y**). Για τον σκοπό αυτό έχει δημιουργηθεί η κλάση **FindLogRelations**, η οποία αποτελείται από δύο μεθόδους

- Την **getDirectAndInverseFollowRelations**: η οποία έχοντας σαν είσοδο ένα Tuple τύπου (Pair, Set[PairNotation]) υπολογίζει αν για το ζεύγος αυτό ισχύει η σχέση ακολουθίας (δηλαδή a>b) και η σχέση μη-ακολουθίας (δηλαδή b>a). Για παράδειγμα η είσοδος αυτής της μεθόδου θα μπορούσε να ήταν η (AB, Set[PairNotation(DIRECT, FOLLOW) , PairNotation(INVERSE, FOLLOW)]) και η έξοδος θα ήταν δύο λίστες με όνομα **directFollow** και **inverseFollow** αντίστοιχα που θα περιέχουν τις σχέσεις αυτές (**κώδικας 12**).
- Την **extractFootPrintGraph**: η οποία δέχεται σαν παραμέτρους το αποτέλεσμα της προηγούμενης μεθόδου και ψάχνει για τις παρακάτω σχέσεις αν υπάρχουν.
  - a->b iff a>b && !(b>a) (Relation.CAUSALITY)
  - a||b iff a>b && b>a (Relation.PARALLEL)
  - a#b iff !(a>b)&& !(b>a) (Relation.NEVER_FOLLOW)

  Συγκεκριμένα όπως φαίνεται στον **κώδικα 12**, ελέγχει τις λίστες **directFollow** και **inverseFollow** και αν και οι δύο δεν είναι άδειες τότε η σχέση μεταξύ των δύο δραστηριοτήτων είναι παραλληλία (Relation.PARALLEL), αν και οι δύο είναι άδειες τότε έχουν σχέση επιλογής δηλαδή Relation.NEVER_FOLLOW στον κώδικα (ποτέ η μια δραστηριότητα δεν ακολουθεί την άλλη και αντίστροφα) και αν η μία από τις δύο δεν είναι άδεια τότε η σχέση των δραστηριοτήτων είναι σχέση «αιτιατότητας» δηλαδή Relation.CAUSALITY στον κώδικα.

```
//Κώδικας 12
def getDirectAndInverseFollowRelations(pairInfo: (Pair, Set[PairNotation])): ((Pair,
Set[PairNotation]), List[PairNotation], List[PairNotation]) = {
  val directFollow = pairInfo._2.toSeq
    .filter(x=> x.getDirectionality()==Directionality.DIRECT &&
x.getRelation()==Relation.FOLLOW)
    .toList

  val inverseFollow = pairInfo._2.toSeq
    .filter(x=> x.getDirectionality()==Directionality.INVERSE &&
x.getRelation()==Relation.FOLLOW)
    .toList

  (pairInfo, directFollow, inverseFollow)
}

def extractFootPrintGraph(pairInfo: (Pair, Set[PairNotation]), directFollow:
List[PairNotation], inverseFollow: List[PairNotation]): (Pair, String) = {
  if (directFollow.nonEmpty && inverseFollow.nonEmpty) {
    (pairInfo._1, Relation.PARALLELISM.toString)
  } else if (directFollow.nonEmpty && inverseFollow.isEmpty) {
    (pairInfo._1, Relation.CAUSALITY.toString)
  } else if (directFollow.isEmpty && inverseFollow.nonEmpty) {
    (new Pair(pairInfo._1.member2, pairInfo._1.member1), Relation.CAUSALITY.toString)
  } else if (directFollow.isEmpty && inverseFollow.isEmpty) {
    (pairInfo._1, Relation.NEVER_FOLLOW.toString)
  } else {
    //this never must happen. Default case
    (pairInfo._1, Relation.FOLLOW.toString)
```

```
    }
}
```

Μετά τον υπολογισμό του πίνακας σχέσεων, ακολουθεί το επόμενο τμήμα του τέταρτου βήματος του αλγορίθμου Alpha, στο οποίο πρέπει να υπολογιστούν τα causal groups. Αυτό υλοποιείται με την κλάση **FindCausalGroups** η οποία δέχεται σαν όρισμα τον πίνακα σχέσεων με την μεταβλητή **logRelations**. Η μεταβλητή αυτή είναι της μορφής Dataset[(Pair, String)] και ένα παράδειγμα των δεδομένων που θα μπορούσε να περιέχει είναι το παρακάτω

(Pair(A, D), CAUSALITY)
(Pair(A, C), CAUSALITY)
(Pair(B, D), CAUSALITY)
(Pair(B, C), CAUSALITY)
(Pair(A, B), NEVER_FOLLOW)
(Pair(C, D), NEVER_FOLLOW)

Σκοπός είναι από τα παραπάνω δεδομένα να βρεθούν δύο σύνολα από δραστηριότητες έστω (Q,R) για τις οποίες κάθε στοιχείο του Q θα ενώνεται με οποιοδήποτε άλλο στοιχείο του R με σχέση «αιτιότητας», ενώ παράλληλα οι δραστηριότητες των Q και R μεταξύ τους θα έχουν σχέση «επιλογής». Για την εύρεση των causal groups ακολουθήθηκαν τα παρακάτω βήματα

1. Εύρεση όλων των μοναδικών δραστηριοτήτων από κάθε πλευρά όλων των causal σχέσεων. Για το παραπάνω παράδειγμα πρέπει να υπολογιστούν δύο σύνολα, ένα με τις δραστηριοτήτες {a,b} και ένα δεύτερο με τις {c,d}.
2. Εύρεση όλων των δυνατών συνδιασμών για κάθε ένα από τα παραπάνω σύνολα. Δηλαδή για το συγκεκριμένο παράδειγμα πρέπει να υπολογιστούν οι δύο λίστες List[{a,b} {a} {b}] and List[{c,d} {c} {d}] .
3. Διαγραφή από τις δύο παραπάνω λίστες των συνόλων αυτών τα οποία έχουν έστω μια σχέση η οποία δεν είναι «επιλογής». Στο συγκεκριμένο παράδειγμα δεν θα γίνει καμία διαγραφή συνόλου.
4. Σύνδεση όλων των συνόλων από τις δύο λιστές μόνο αν όλα τα στοιχεία των δύο λιστών είναι σε σχέση «αιτιατότητας» μεταξύ τους και σχηματισμός των causal groups.

Στον κώδικα 13 φαίνονται τα παραπάνω βήματα για τον υπολογισμό των causal groups.

```
//Κώδικας 13
def extractCausalGroups():Dataset[CausalGroup[String]] = {
  val directCausalRelations = logRelations
    .filter(x=>x._2==Relation.CAUSALITY.toString)
    .map(x=>x._1)
    .map(x=>(x.member1, x.member2))

  //1) Find all events from both sides (left and right) {a,b} and {c,d}
  val uniqueEventsFromLeftSideEvents = directCausalRelations
    .map(causal => ("left", causal._1))
```

```
    .groupByKey(x=>x._1)
    .mapGroups{case(k, iter) => (k, iter.map(x => x._2).toSet)}

  val uniqueEventsFromRightSideEvents = directCausalRelations
    .map(causal => ("right", causal._2))
    .groupByKey(x=>x. 1)
    .mapGroups{case(k, iter) => (k, iter.map(x => x._2).toSet)}

  //2) Find all possible combination for each set (left and right) List[{a,b} {a} {b}] and
List[{c,d} {c} {d}]
  //3) Delete from above lists all the sets which are not in never follow relation with each
other
  val groups = uniqueEventsFromLeftSideEvents
    .union(uniqueEventsFromRightSideEvents)
    .flatMap(uniqueEvents => possibleSubsets(uniqueEvents))
    .filter(subset=> allRelationsAreNeverFollow(subset._2))
    .groupByKey(x=>x. 1)
    .mapGroups{case(k, iter) => (k, iter.map(x => x._2).toList)}

  import spark.implicits._
  val leftGroups = groups.first()
  val rigthGroups = groups.orderBy($"value".desc).first()

  //4) Connect all the sets (as causal groups) from two lists and keep only those for which
all events are in causal relations
  val causalGroups = leftGroups._2.toDS().as("left")
    .crossJoin(rigthGroups._2.toDS().as("right"))
    .where($"left.value" !== $"right.value")
    .toDF("left", "right")
    .filter(group=>isCausalRelationValid(group.getAs[Seq[String]]("left").toSet,
group.getAs[Seq[String]]("right").toSet))
    .map(group=>new
CausalGroup(group.getAs[Seq[String]]("left").toSet,group.getAs[Seq[String]]("right").toSet))

  causalGroups
}
```

Για να βρεθεί αν ένα ζεύγος από δραστηριότητες ανήκει σε σχέση «επιλογής» ή σε σχέση «αιτιατότητας» χρησιμοποιήθηκαν broadcast μεταβλητές οι οποίες είναι κοινόχρηστες μεταβλητές (shared variables) και χρησιμοποιούνται για να επιτρέψουν στον προγραμματιστή να κρατήσει ένα αντίγραφο μόνο για ανάγνωση των δεδομένων σε κάθε κόμβο της συστάδας υπολογιστών αντί να στέλνει τα δεδομένα μαζί με τις διάφορες εργασίες που πρέπει να εκτελεστούν. Έτσι με αυτόν τον τρόπο μπορεί κάθε κόμβος να ελέγξει πολύ γρήγορα αν ένα ζεύγος από δραστηριότητες ανήκει σε κάποια από τις δύο σχέσεις. Παράδειγμα η μέθοδος **allRelationsAreNeverFollow** στον **Κώδικας 14** που ελέγχει αν όλα τα πιθανά ζεύγη ενός σύνολου από δραστηριότητες ανήκουν στην σχέση επιλογής χρησιμοποιώντας μεταβλητές broadcast. Ίδια ακριβώς χρήση των μεταβλητών broadcast κάνει και η μέθοδος **allEventsAreInCausalityRelation** για να ελέγξει αν όλα τα στοιχεία δύο συνόλων από δραστηριότητες είναι σε σχέση «αιτιατότητας» μεταξύ τους.

```
//Κώδικας 14
val never = logRelations
  .filter(x=>x._2==Relation.NEVER_FOLLOW.toString)
  .map(neverRelation => neverRelation. 1)
  .distinct()
  .collect()
  .toList

val neverBc = spark.sparkContext.broadcast(never)

val causal = logRelations
  .filter(x=>x._2==Relation.CAUSALITY.toString)
  .map(neverRelation => neverRelation._1)
  .distinct()
```

```
  .collect()
  .toList

val causalBc = spark.sparkContext.broadcast(causal)

def allRelationsAreNeverFollow(possibleGroup: Set[String]): Boolean = {
  val allPossiblePairs = for {
    (x, idxX) <- possibleGroup.zipWithIndex
    (y, idxY) <- possibleGroup.zipWithIndex
    if idxX < idxY
  } yield new Pair(x,y)
  val notNeverFollow = allPossiblePairs
    .find(x=> (!neverBc.value.contains(x) && !neverBc.value.contains(createInversePair(x))))

  notNeverFollow.isEmpty
}

def allEventsAreInCausalityRelation(groupA: Set[String], groupB: Set[String]): Boolean = {
  val pairs = for {
    eventA <- groupA
    eventB <- groupB
  } yield new Pair(eventA, eventB)

  for {
    pair <- pairs
  } yield if (!causalBc.value.contains(pair)) { return false }

  true
}
```

Το πέμπτο βήμα του αλγόριθμου Alpha είναι ο υπολογισμός μόνο των μέγιστων causal groups. Δηλαδή από τα causal groups που υπολογίστηκαν στο προηγούμενο βήμα πρέπει να διατηρηθούν για την συνέχεια της εκτέλεσης του αλγορίθμου μόνο αυτά τα causal groups που είναι μέγιστα, δηλαδή causal groups που δεν συμπεριλαμβάνονται σε άλλα. Για παράδειγμα έστω ότι υπάρχουν 2 causal groups ({a} , {b}) και ({a} , {b,c}). Από αυτά τα causal groups μόνο το δεύτερο ({a} , {b,c}) πρέπει να διατηρηθεί γιατί είναι μέγιστο καθώς περιέχει και το ({a} , {b}). Αυτό το βήμα του αλγόριθμου **Alpha** εκτελείται στην κλάση **FindMaximalPairs** (**Κώδικας 15**). Συγκεκριμένα γίνεται χρήση των συσσωρευτών (**accumulators**) του Spark αλλά και των μεταβλητών broadcast. Οι συσσωρευτές είναι μεταβλητές που μπορούν να χρησιμοποιηθούν σαν χώρος για την συσσώρευση μερικών αποτελεσμάτων που εκτελούνται σε διαφορετικούς κόμους της συστάδας υπολογιστών.

Η διαδικασία εύρεσης των maximal groups είναι αρκετά απλή και για κάθε causal group στο αρχικό σύνολο ελέγχεται αν υπάρχει κάποιο άλλο group που να είναι υποσύνολο του (**isSubsetOf** μέθοδος). Αν δεν βρεθεί κανένα τότε το causal group προστίθεται στον συσσωρευτή (maximalCausalGroups.add(group)).

```
//Κώδικας 15
val spark = SparkSession.builder().getOrCreate()
val causalGroupsList = causalGroups
  .distinct()
  .collect()
  .toList

val causalGroupsListBc = spark.sparkContext.broadcast(causalGroupsList)

def extract(): List[CausalGroup[String]] = {
  val maximalCausalGroups : CollectionAccumulator[CausalGroup[String]] =
spark.sparkContext.collectionAccumulator("maximalCausalGroups")

  causalGroups
    .foreach(group => if (toBeRetained(group)) {
```

```
      maximalCausalGroups.add(group)
    })

  maximalCausalGroups.value.asScala.toList
}

def toBeRetained(toCheck: CausalGroup[String]): Boolean = {
  !causalGroupsListBc.value.exists(x => x != toCheck && isSubsetOf(toCheck, x))
}

/**
  * If true, the group2 must be retained
  * @param group1
  * @param group2
  * @return
  */
def isSubsetOf(group1 : (CausalGroup[String]), group2 : CausalGroup[String]) : Boolean = {
  return group1.getFirstGroup().subsetOf(group2.getFirstGroup()) &&
group1.getSecondGroup().subsetOf(group2.getSecondGroup())
}
```

Το έκτο βήμα του αλγορίθμου είναι η εύρεση των καταστάσεων του παραγόμενου από τον αλγόριθμο **Alpha** δίκτυο Petri. Η αναπαράσταση μιας κατάστασης του δικτύου Petri γίνεται με την κλάση State η οποία διαθέτει δύο παραμέτρους όπως φαίνεται στην αρχή του **κώδικα 16**. Η μία παράμετρος είναι η input και είναι ένα σύνολο από δεδομένα τύπου String, καθώς μια κατάσταση μπορεί να έχει σαν είσοδο παραπάνω από μια δραστηριότητες. Ομοίως και για την έξοδο του State. Έτσι λοιπόν τα μέγιστα causal groups που υπολογίστηκαν στο προηγούμενο βήμα μετατρέπονται σε καταστάσεις του δίκτυου Petri, όπως φαίνεται στον **κώδικα 16**. Ακόμα δημιουργούνται και δύο επιπλέον καταστάσεις, μια η αρχική κατάσταση και μια η τελική κατάσταση του δίκτυου Petri. Οι κλάση Places συνοψίζει τις καταστάσεις του δίκτυου Petri σε ένα tuple με τρία στοιχεία.

```
//Κώδικας 16
@SerialVersionUID(100L)
class State(val input: Set[String], val output: Set[String]) extends Serializable {
}

@SerialVersionUID(100L)
class Places(val places: (State, State, List[State])) extends Serializable {
}

def getPlaces(maximalGroups : List[CausalGroup[String]], startActivities : Set[String],
finalActivities : Set[String]): Places = {
  val states = maximalGroups
    .map(x=> new State(x.getFirstGroup(), x.getSecondGroup()))

  val initialState = new State(Set.empty, startActivities)
  val finalState = new State(finalActivities, Set.empty)

  new Places(initialState, finalState, states)
}
```

Το έβδομο βήμα του αλγόριθμου **Alpha** είναι ο υπολογισμός των τόξων (arcs) του δικτύου Petri. Η υλοποίηση αυτού του βήματος γίνεται στην κλάση FindEdges, όπως φαίνεται στον **κώδικα 17**. Η κλάση αυτή δέχεται σαν παράμετρο ένα στιγμιότυπο της κλάσης Places το οποίο υπολογίστηκε στο προηγούμενο βήμα. Σκοπός της είναι να παράγει μια λίστα από στιγμιότυπα της κλάσης Edge, που θα αναπαριστούν τα τόξα του γραφήματος Petri. Κάθε στιγμιότυπο της κλάσης Edge αποτελείται από τρεις παραμέτρους μια δραστηριότητα, μια

κατάσταση και μια Boolean μεταβλητή που δείχνει ποιο από τα δύο είναι η αρχή και ποιο το πέρας του τόξου. Αν η Boolean τιμή είναι true, τότε η δραστηριότητα είναι η αρχή του τόξου και η κατάσταση το πέρας. Αν είναι false, τότε συμβαίνει το αντίθετο. Η κατασκευή των τόξων γίνεται στην μέθοδο **constructEdgesFromStates** η οποία δέχεται σαν είσοδο μια κατάσταση και μετατρέπει τις εισόδους της κατάστασης σε στιγμιότυπο της κλάσης Edge με boolean true (καθώς η κατάσταση είναι το πέρας του τόξου) και τις εξόδους της κατάστασης σε στιγμιότυπο της κλάσης Edge με boolean false (καθώς η κατάσταση είναι η αρχή του τόξου). Ακόμα κατασκευάζει τα στιγμιότυπα της κλάσης Edge που ξεκινάνε από την αρχική κατάσταση με την μέθοδο **getInitialEdges** και τα στιγμιότυπα της κλάσης Edge που καταλήγουν στην τελική κατάσταση με την μέθοδο **getFinalEdges**.

//Κώδικας 17

```
/**
  * Direct == true -> (event, State)
  * Inverse == false -> (State, event)
  */
@SerialVersionUID(100L)
class Edge(val event: String, val state: State, val direct: Boolean)  extends Serializable {}


@SerialVersionUID(100L)
class FindEdges(val places: Places) extends Serializable {

  def find():List[Edge] = {
    val edges = places.getStates()
      .flatMap(x=>constructEdgesFromStates(x))

    return edges ::: getInitialEdges() ::: getFinalEdges()
  }

  def constructEdgesFromStates(state : State) : List[Edge] = {
    val inputEdges = state.getInput()
      .map(x=> new Edge(x, state, true))
      .toList

    val outputEdges = state.getOutput()
      .map(x=> new Edge(x, state, false))
      .toList

    return inputEdges ::: outputEdges
  }

  def getInitialEdges(): List[Edge] = {
    return places.getInitialState().getOutput()
      .map(x=> new Edge(x, places.getInitialState(), false))
      .toList
  }

  def getFinalEdges(): List[Edge] = {
    return places.getFinalState().getInput()
      .map(x=> new Edge(x, places.getFinalState(), true))
      .toList
  }

}
```

Το όγδωο και τελευταίο βήμα του αλγόριθμου **Alpha** είναι η κατασκευή του τελικού δικτύου Petri, που είναι η δημιουργία ενός στιγμιοτύπου της κλάσης **PetriNet** (**κώδικας 18**), η οποία διαθέτει τρεις παραμέτρους μια για τις καταστάσεις (places), μια για τις δραστηριότητες (events) και μία για τα τόξα (edges) του δικτύου Petri.

```
//Κώδικας 18
@SerialVersionUID(100L)
class PetriNet (val places: Places, val events: List[String], val edges: List[Edge]) extends
Serializable {}
```

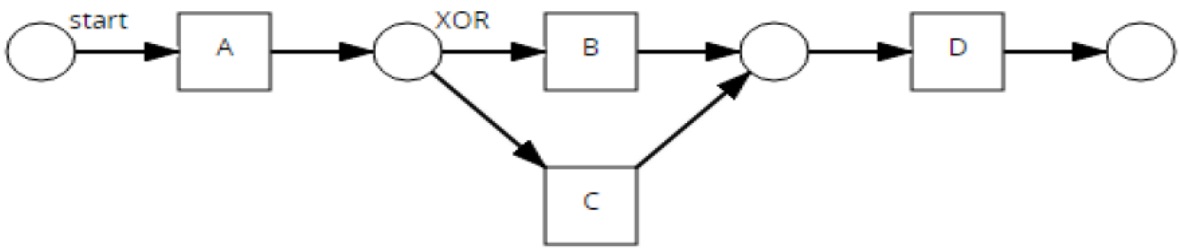
Ο ίδιος ακριβώς αλγόριθμος υλοποιήθηκε και δεύτερη φορά χωρίς την χρήση **Spark** βιβλιοθηκών ώστε να γίνει σύγκριση στο ίδιο περιβάλλον εκτέλεσης του αλγόριθμοy Alpha σε **Spark** περιβάλλον αλλά και σε ένα περιβάλλον χωρίς την χρήση του **Spark**.

5.2 Επιβεβαίωση Ορθής Λειτουργίας του Κώδικα Υλοποίησης

Για την επαλήθευση της ορθής λειτουργίας του υλοποιημένου αλγόριθμου Alpha έχουν γραφτεί συνολικά 34 προγράμματα δοκιμής (unit tests), τα οποία ελέγχουν την λειτουργία του αλγορίθμου από την αρχή έως το τέλος αλλά και τις κάθε κλάσης ξεχωριστά. Αυτό επιτρέπει την αλλαγή της υλοποίησης σε διάφορα τμήματα του αλγορίθμου γνωρίζοντας ότι πάντα η συνολική λειτουργικότητα του αλγορίθμου μένει ανεπηρέαστη.

Για τον από άκρη σε άκρη έλεγχο χρησιμοποιήθηκαν 5 παραδείγματα που είναι γνωστό εκ των προτέρων το τελικό δίκτυο Petri το οποίο πρέπει να παραχθεί. Αντίστοιχα έγινε η υλοποίηση των αντίστοιχων προγραμμάτων δοκιμής τα οποία περιμένουν το ίδιο ακριβώς δίκτυο Petri να παραχθεί και από την **Spark** υλοποίηση του αλγόριθμου Alpha. Παρακάτω παρουσιάζονται σύντομα τα 5 παραδείγματα και τα αντίστοιχα προγράμματα δοκιμής.

Παράδειγμα 1



Εικόνα 37:
Το δίκτυο Petri που παράγεται από το σύνολο γεγονότων w = {ABD , ACD}

Στον **κώδικα 19** φαίνεται το αντίστοιχο προγράμμα δοκιμής σε Scala που υλοποιήθηκε και επαληθεύει την ορθή λειτουργία του υλοποιημένου αλγόριθμου Alpha. Συγκεκριμένα από τον παρακάτω κώδικα βλέπουμε ότι υπάρχουν 4 διαφορετικές δραστηριότητες (A,B,C,D) όσα ακριβώς υπάρχουν και στο δίκτυο Petri. Ακόμα βλέπουμε ότι ο αριθμός των τόξων στο

δίκτυο Petri είναι 8, όσα ακριβώς τόξα βρέθηκαν και από τον υλοποιημένο αλγόριθμο Alpha (τελευταίο assertion). Ακόμα στο δίκτυο Petri είναι φανερό ότι υπάρχουν τέσσερεις καταστάσεις (κύκλοι πάνω στο γράφημα). Ακριβώς τις ίδιες καταστάσεις έχουν βρεθεί και από τον υλοποιημένο αλγόριθμο Alpha και είναι οι ακόλουθες

- State(*Set*.empty, *Set*("A")) : Η αρχική κατάσταση του δικτύου Petri (start) που δεν έχει είσοδο και σαν έξοδο έχει την δραστηριότητα A.
- State(*Set*("A"), *Set*("B", "C")) : Την δεύτερη κατάσταση του δικτύου Petri που έχει είσοδο την δραστηριότητα A και σαν έξοδο έχει τις δραστηριότητες B και C.
- State(*Set*("B", "C"), *Set*("D")): Την τρίτη κατάσταση του δικτύου Petri που έχει είσοδο τις δραστηριότητες B και C και σαν έξοδο έχει τη δραστηριότητα D.
- State(*Set*("D"), *Set*.empty): Την τέταρτη κατάσταση του δικτύου Petri είναι η τελική και έχει σαν είσοδο τη δραστηριότητα D ενώ δεν έχει καμία έξοδο.

Ακόμα από το δίκτυο Petri και το πρόγραμμα δοκιμής βλέπουμε ότι ο υλοποιημένος αλγόριθμος Alpha έχει βρει και τις σωστές ακμές οι οποίες είναι 8 και είναι οι ακόλουθες

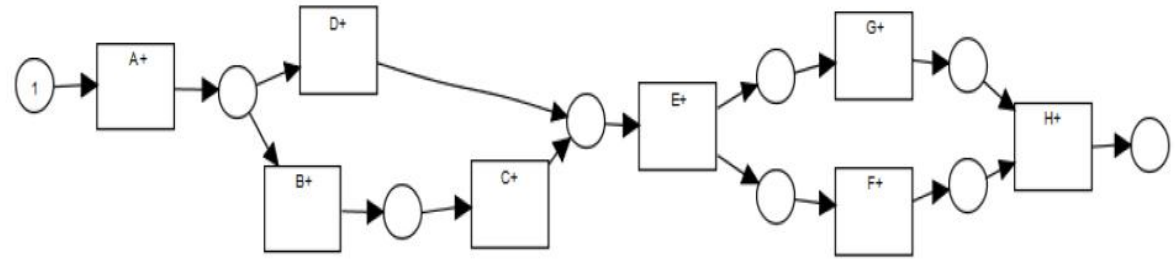
- Edge("A", **new** State(*Set*("A"), *Set*("B", "C")), **true**)) : Ενώνει την δραστηριότητα A με τη δεύτερη κατάσταση του δικτύου Petri. Το boolean flag είναι true γιατί η δραστηριότητα είναι η αρχή της ακμής και το κατάσταση είναι το πέρας της.
- Edge("B", **new** State(*Set*("A"), *Set*("B", "C")), **false**)) : Ενώνει την δεύτερη κατάσταση του δικτύου Petri με τη δραστηριότητα B. Το boolean flag είναι false γιατί η κατάσταση είναι η αρχή της ακμής και η δραστηριότητα είναι το πέρας της.
- Edge("C", **new** State(*Set*("A"), *Set*("B", "C")), **false**)) : Ενώνει τη δεύτερη κατάσταση του δικτύου Petri με τη δραστηριότητα C. Το boolean flag είναι false γιατί η κατάσταση είναι η αρχή της ακμής και η δραστηριότητα είναι το πέρας της.
- Edge("B", **new** State(*Set*("B", "C"), *Set*("D")), **true**)) : Ενώνει τη δραστηριότητα B με την τρίτη κατάσταση του δικτύου Petri. Το boolean flag είναι true γιατί η δραστηριότητα είναι η αρχή της ακμής και η κατάσταση είναι το πέρας της.
- Edge("C", **new** State(*Set*("B", "C"), *Set*("D")), **true**)) : Ενώνει τη δραστηριότητα C με τη τρίτη κατάσταση του δικτύου Petri. Το boolean flag είναι true γιατί η δραστηριότητα είναι η αρχή της ακμής και η κατάσταση είναι το πέρας της.
- Edge("D", **new** State(*Set*("B", "C"), *Set*("D")), **false**)) : Ενώνει τη τρίτη κατάσταση του δικτύου Petri με τη δραστηριότητα D. Το boolean flag είναι false γιατί η κατάσταση είναι η αρχή της ακμής και η δραστηριότητα είναι το πέρας της.
- Edge("A", **new** State(*Set*.empty, *Set*("A")), **false**)) : Ενώνει την πρώτο κατάσταση του δικτύου Petri με την δραστηριότητα A. Το boolean flag είναι false γιατί η κατάσταση είναι η αρχή της ακμής και η δραστηριότητα είναι το πέρας της.
- Edge("D", **new** State(*Set*("D"), *Set*.empty), **true**)) : Ενώνει τη δραστηριότητα D με την τελευταία κατάσταση του δικτύου Petri. Το boolean flag είναι true γιατί η δραστηριότητα είναι η αρχή της ακμής και η κατάσταση είναι το πέρας της.

Ακόμα στον κώδικα 19 υπάρχουν και δύο επιβεβαιώσεις που ελέγχουν συνθήκες οι οποίες δεν πρέπει να ισχύουν.

//Κώδικας 19

```
test("Check Alpha Algorithm functionality - Log 2") {
  val logPath = "src/test/resources/log2.txt"
  val traceTools: TraceTools = new TraceTools()
  val tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)
  val petriNet = AlphaAlgorithm.executeAlphaAlgorithm(tracesDS)

  //check edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A"), Set("B", "C")), true)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("A"), Set("B", "C")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("A"), Set("B", "C")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("B", "C"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), false)))

  //check initial edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set.empty, Set("A")), false)))

  //check final edges
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("D"), Set.empty), true)))

  //check wrong directionality
  assert(!petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), false)))
  assert(!petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), true)))

  assert(petriNet.getEdges().size==8)
}
```

Παράδειγμα 2



Εικόνα 38:
Το Petri Net παράγεται από το event log w = {ABCEFGH , ABCEGFH, ADEGFH, ADEFGH }

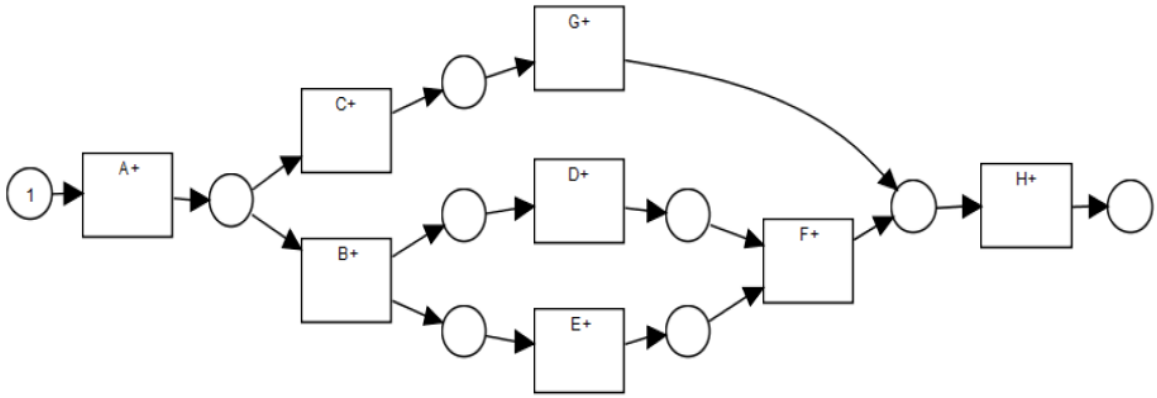
Ο κώδικας 20 ελέγχει την ορθή λειτουργία του αλγόριθμου **Alpha** για ένα σύνολο γεγονότων w = {ABCEFGH , ABCEGFH, ADEGFH, ADEFGH }. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος αλγόριθμος **Alpha** κατασκευάζει το σωστό δίκτυο Petri με ακριβώς 18 ακμές και 9 καταστάσεις.

//Κώδικας 20

```
test("Check Alpha Algorithm functionality - Log 5") {
  val logPath = "src/test/resources/log5.txt"
  val traceTools: TraceTools = new TraceTools()
  val tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)
  val petriNet = AlphaAlgorithm.executeAlphaAlgorithm(tracesDS)

  //check edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A"), Set("B","D")), true)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("A"), Set("B","D")), false)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("A"), Set("B","D")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B"), Set("C")), true)))
```

```
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("B"), Set("C")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("C","D"), Set("E")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("C","D"), Set("E")), true)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("C","D"), Set("E")), false)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("E"), Set("F")), true)))
  assert(petriNet.getEdges().contains(new Edge("F", new State(Set("E"), Set("F")), false)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("E"), Set("G")), true)))
  assert(petriNet.getEdges().contains(new Edge("G", new State(Set("E"), Set("G")), false)))
  assert(petriNet.getEdges().contains(new Edge("F", new State(Set("F"), Set("H")), true)))
  assert(petriNet.getEdges().contains(new Edge("H", new State(Set("F"), Set("H")), false)))
  assert(petriNet.getEdges().contains(new Edge("G", new State(Set("G"), Set("H")), true)))
  assert(petriNet.getEdges().contains(new Edge("H", new State(Set("G"), Set("H")), false)))

  //check initial edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set.empty, Set("A")), false)))

  //check final edges
  assert(petriNet.getEdges().contains(new Edge("H", new State(Set("H"), Set.empty), true)))

  //check wrong directionality
  assert(!petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), false)))
  assert(!petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), true)))

  assert(petriNet.getEdges().size==18)
}
```

Παράδειγμα 3



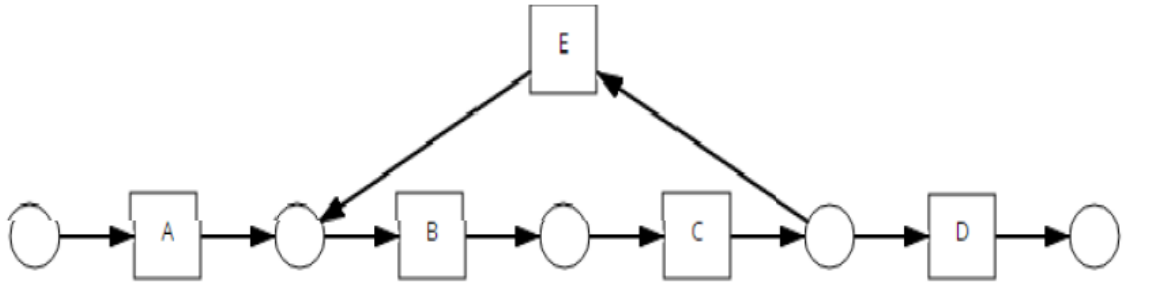
Εικόνα 39:
Το Petri Net παράγεται από το event log w = {ABDEFH , ACGH, ABEDFH }

Ο κώδικας 21 ελέγχει την ορθή λειτουργία του αλγόριθμου **Alpha** για ένα σύνολο γεγονότων w = {ABDEFH , ACGH, ABEDFH }. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος αλγόριθμος **Alpha** κατασκευάζει το σωστό δίκτυο Petri με ακριβώς 18 ακμές και 9 καταστάσεις.

```
//Κώδικας 21
test("Check Alpha Algorithm functionality - Log 3") {
  val logPath = "src/test/resources/log3.txt"
  val traceTools: TraceTools = new TraceTools()
  val tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)
  val petriNet = AlphaAlgorithm.executeAlphaAlgorithm(tracesDS)

  //check edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A"), Set("B", "C")), true)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("A"), Set("B", "C")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("A"), Set("B", "C")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("C"), Set("G")), true)))
  assert(petriNet.getEdges().contains(new Edge("G", new State(Set("C"), Set("G")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("B"), Set("D")), false)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("D"), Set("F")), true)))
  assert(petriNet.getEdges().contains(new Edge("F", new State(Set("D"), Set("F")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B"), Set("E")), true)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("B"), Set("E")), false)))
```

```
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("E"), Set("F")), true)))
  assert(petriNet.getEdges().contains(new Edge("F", new State(Set("E"), Set("F")), false)))
  assert(petriNet.getEdges().contains(new Edge("F", new State(Set("F", "G"), Set("H")), true)))
  assert(petriNet.getEdges().contains(new Edge("G", new State(Set("F", "G"), Set("H")), true)))
  assert(petriNet.getEdges().contains(new Edge("H", new State(Set("F", "G"), Set("H")), false)))

  //check initial edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set.empty, Set("A")), false)))

  //check final edges
  assert(petriNet.getEdges().contains(new Edge("H", new State(Set("H"), Set.empty), true)))

  //check wrong directionality
  assert(!petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), false)))
  assert(!petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), true)))

  assert(petriNet.getEdges().size==18)
}
```

Παράδειγμα 4



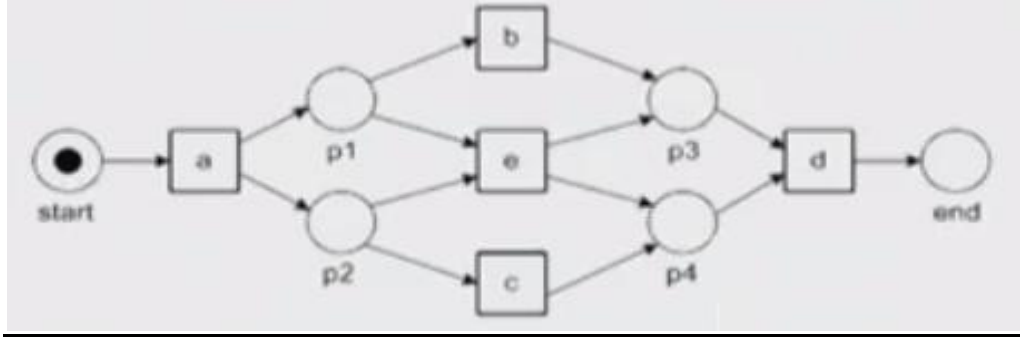
Εικόνα 40:
Το Petri Net παράγεται από το event log w = {ABCEBCD , ABCD}

Ο κώδικας 22 ελέγχει την ορθή λειτουργία του αλγόριθμου **Alpha** για ένα σύνολο γεγονότων w = {ABCEBCD , ABCD}. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος αλγόριθμος **Alpha** κατασκευάζει το σωστό δίκτυο Petri με ακριβώς 10 ακμές και 5 καταστάσεις και επιπλέον διαθέτει και μια επανάληψη (loop).

```
//Κώδικας 22
test("Check Alpha Algorithm functionality - Log 4") {
  val logPath = "src/test/resources/log4.txt"
  val traceTools: TraceTools = new TraceTools()
  val tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)
  val petriNet = AlphaAlgorithm.executeAlphaAlgorithm(tracesDS)

  //check edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A", "E"), Set("B")), true)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("A", "E"), Set("B")), true)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("A", "E"), Set("B")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B"), Set("C")), true)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("B"), Set("C")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("C"), Set("D", "E")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("C"), Set("D", "E")), false)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("C"), Set("D", "E")), false)))

  //check initial edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set.empty, Set("A")), false)))

  //check final edges
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("D"), Set.empty), true)))

  //check wrong directionality
```

```
  assert(!petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), false)))
  assert(!petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), true)))

  assert(petriNet.getEdges().size==10)
}
```

Παράδειγμα 5



Εικόνα 41:
Το Petri Net παράγεται από το event log w = {ABCD , ACBD, AED}

Ο κώδικας 22 ελέγχει την ορθή λειτουργία του αλγόριθμου **Alpha** για ένα σύνολο γεγονότων w = { ABCD , ACBD, AED}. Παρατηρούμε, ακριβώς με την ίδια ανάλυση του παραδείγματος 1, ότι ο υλοποιημένος αλγόριθμος **Alpha** κατασκευάζει το σωστό δίκτυο Petri με ακριβώς 14 ακμές και 6 καταστάσεις και επιπλέον διαθέτει και μια επανάληψη (loop).

```
//Κώδικας 23
test("Check Alpha Algorithm functionality - Log 1") {
  val logPath = "src/test/resources/log1.txt"
  val traceTools: TraceTools = new TraceTools()
  val tracesDS : Dataset[(String, List[String])] = traceTools.tracesDSFromLogFile(logPath)
  val petriNet = AlphaAlgorithm.executeAlphaAlgorithm(tracesDS)

  //check edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A"), Set("B", "E")), true)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("A"), Set("B", "E")), false)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("A"), Set("B", "E")), false)))
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set("A"), Set("C", "E")), true)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("A"), Set("C", "E")), false)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("A"), Set("C", "E")), false)))
  assert(petriNet.getEdges().contains(new Edge("B", new State(Set("B", "E"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("B", "E"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("B", "E"), Set("D")), false)))
  assert(petriNet.getEdges().contains(new Edge("C", new State(Set("C", "E"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("E", new State(Set("C", "E"), Set("D")), true)))
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("C", "E"), Set("D")), false)))

  //check initial edges
  assert(petriNet.getEdges().contains(new Edge("A", new State(Set.empty, Set("A")), false)))

  //check final edges
  assert(petriNet.getEdges().contains(new Edge("D", new State(Set("D"), Set.empty), true)))

  //check wrong directionality
  assert(!petriNet.getEdges().contains(new Edge("B", new State(Set("B", "C"), Set("D")), false)))
  assert(!petriNet.getEdges().contains(new Edge("D", new State(Set("B", "C"), Set("D")), true)))

  assert(petriNet.getEdges().size==14)
}
```

6. Πειραματική αξιολόγηση

Στο κεφάλαιο αυτό παρουσιάζονται το υπολογιστικό περιβάλλον στο οποίο έγιναν τα πειράματα του υλοποιημένου αλγόριθμου Alpha, τα δεδομένα που χρησιμοποιήθηκαν σαν είσοδο στον αλγόριθμο Alpha αλλά και η ανάλυση των πειραματικών αποτελεσμάτων.

6.1 Υπολογιστικό περιβάλλον

Για την πειραματική αξιολόγηση του αλγόριθμου Alpha έγιναν μετρήσεις στην πλατφόρμα Databricks η οποία παρέχει δημιουργία συστάδας υπολογιστών με μεταβλητό αριθμό από κόμβους. Η **Databricks (https://databricks.com/)** είναι μια εταιρεία η οποία ιδρύθηκε από τους δημιουργούς του **Apache Spark** και σκοπεύει να βοηθήσει τους πελάτες που χρειάζονται ανάλυση Μεγάλων Δεδομένων βασιζόμενοι στο υπολογιστικό νέφος (cloud) χρησιμοποιώντας το **Spark**. Το **Databricks** είναι μια πλατφόρμα που δουλεύει με το **Spark**, και παρέχει αυτόματη διαχείρηση μιας συστάδας υπολογιστών και διάφορα περιβάλλοντα συγγραφής κώδικα (notebooks), τα οποία παρέχουν πολύ εύκολη συγγραφή και εκτέλεση **Spark** κώδικα στην συστάδα που έχει δημιουργήσει ο χρήστης.

Για την μέτρηση και αξιολόγηση του αλγόριθμου **Alpha** με την χρήση του Spark, αλλά και χωρίς αυτήν, χρησιμοποιήθηκε η **Community edition** του **Databricks**, η οποία παρέχει στον χρήστη ένα cluster με μόνο ένα πρόγραμμα οδηγό (**driver** αλλά χωρίς κόμβους) το οποίο διαθέτει 6 GB μνήμης RAM. Ακόμα χρησιμοποιήθηκε και η κανονική έκδοση του Databricks, η οποία στην συγκεκριμένη έκδοση που έγιναν οι μετρήσεις, παρέχει ένα πρόγραμμα οδηγό και 0 έως 4 κόμβους με 14 GB μνήμης RAM ο καθένας.

Για την εκτέλεση του αλγόριθμου **Alpha** δημιουργήθηκαν στο **Databricks**, δύο περιβάλλοντα συγγραφής κώδικα (notebooks). Το πρώτο περιβάλλον εκτελεί τον αλγόριθμο **Alpha** στην **Spark** έκδοση του, ενώ το δεύτερο εκτελεί τον αλγόριθμου **Alpha** χωρίς την χρήση **Spark** βιβλιοθηκών. Στις παρακάτω δύο εικόνες (42 έως 44) φαίνεται το περιβάλλοντα συγγραφής κώδικα (notebook) που δημιουργήθηκε στο **Databricks** για την εκτέλεση του αλγόριθμου **Alpha** με την χρήση **Spark** βιβλιοθηκών.

Cmd 1

```
import tools.TraceTools
import org.apache.spark.sql.{Dataset, SparkSession}
import steps.AlphaAlgorithmSteps
import misc.{CausalGroup, Pair}
import petriNet.PetriNet
import petriNet.flow.Edge
import petriNet.state.Places
import steps.PossibleCombinations
import java.util.concurrent.TimeUnit

val traceTools: TraceTools = new TraceTools()
val logPath = "/FileStore/tables/readDataFiltered.csv"
val numOfTraces = 15
val percentage : Float = 0.5f //delete trace occurrences which are less than 1% from all traces
val readAll : Boolean = true
val filtering : Boolean = true

val tracesDS : Dataset[(String, List[String])] = traceTools.
  getTracesToInspect(logPath, numOfTraces, readAll, filtering, percentage)
```

Cmd 2

```
val steps : AlphaAlgorithmSteps = new AlphaAlgorithmSteps()

//Step 1 - Find all transitions / events, Sorted list of all event types
val events = steps.getAllEvents(tracesDS)
events.length
```

Εικόνα 42:
Notebook στο **Databricks** που τρέχει σε δύο εντολές την δημιουργία των ιχνών δραστηριοτήτων που θα επεξεργαστεί ο αλγόριθμος Alpha και το πρώτο βήμα του αλγορίθμου.

Cmd 3

```
//Step 2 - Construct a set with all start activities (Ti)
val startActivities = steps.getStartActivities(tracesDS)
```

Cmd 4

```
//Step 3 - Construct a set with all final activities (To)
val finalActivities = steps.getFinalActivities(tracesDS)
```

Cmd 5

```
//Step 4 - Footprint graph - Causal groups
val logRelations : Dataset[(Pair, String)] = steps.getFootprintGraph(tracesDS, events)

```

Cmd 6

```
val causalGroups : Dataset[CausalGroup[String]] = steps.getCausalGroups(logRelations)
causalGroups.count()
```

Cmd 7

```
//Step 5 - compute only maximal groups
val maximalGroups : List[CausalGroup[String]] = steps.getMaximalGroups(causalGroups)
```

Εικόνα 43:
Notebook στο **Databricks** με τα βήματα 3 έως 5 του αλγόριθμου Alpha.

Cmd 8

```
//step 6 - set of places/states
val places : Places = steps.getPlaces(maximalGroups, startActivities, finalActivities)
```

Cmd 9

```
//step 7 - set of arcs (flow)
val edges : List[Edge] = steps.getEdges(places)
```

Cmd 10

```
//step 8 - construct petri net
val petriNet: PetriNet = new PetriNet(places, events, edges)

```

Εικόνα 44:
Notebook στο **Databricks** με τα βήματα 6 έως 8 του αλγόριθμου Alpha.

6.2 Δεδομένα πειραματικής αξιολόγησης

Στο κεφάλαιο αυτό θα γίνει παρουσίαση των δεδομένων που χρησιμοποιήθηκαν για να επεξεργαστούν από τον αλγόριθμο Alpha που υλοποιήθηκε. Το σύνολο δεδομένων έχει συγκεντρωθεί από έναν μεγάλο τηλεπικοινωνιακό οργανισμό και είναι ένα csv αρχείο μεγέθους 400mb περίπου και περιέχει συνολικά 36100 ίχνη δραστηριοτήτων (traces) τα οποία πρέπει να επεξεργαστούν από τον αλγόριθμο Alpha. Το csv αρχείο περιέχει πολλές στήλες, όμως για την εκτέλεση του αλγόριθμου Alpha απαραίτητες είναι μόνο οι στήλες «orderId», «eventname», «status» και «starttime». Η στήλη «eventname» περιέχει τις δραστηριότητες που συμβαίνουν και το «orderId» ομαδοποιεί αυτές τις δραστηριότητες σε ίχνη τα οποία θα επεξεργαστούν από τον αλγόριθμο Alpha. Δηλαδή το «orderId» προσδιορίζει μοναδικά κάθε ίχνος δραστηριοτήτων. Η στήλη «status» περιέχει πληροφορία σχετικά με το αν έχει ολοκληρωθεί ή όχι κάποια δραστηριότητα και η στήλη «starttime» περιέχει χρονική πληροφορία σχετικά με το πότε ξεκίνησε η συγκεκριμένη δραστηριότητα ώστε να ταξινομηθεί σώστα μέσα στο ίχνος δραστηριοτήτων που θα εξεταστεί από τον αλγόριθμο.

6.3 Πειραματικά αποτελέσματα

Στο κεφάλαιο αυτό θα γίνει παρουσίαση των μετρήσεων που έγιναν στο **Databricks**, τόσο στην **Community Edition** αλλά και στην κανονική έκδοση που παρέχει συστάδα υπολογιστών με διαφορετικό αριθμό από κόμβους. Οι μετρήσεις έγιναν πάνω στο ίδιο σύνολο δεδομένων αλλά με διαφορετικές παραμέτρους κάθε φορά ώστε στον αλγόριθμο **Alpha** να τροφοδοτείται διαφορετικός αριθμός από ίχνη δραστηριοτήτων και διαφορετικός αριθμός συνολικών δραστηριοτήτων. Σε κάθε πίνακα παρουσιάζονται οι τέσσερεις (4) παρακάτω χρόνοι

- Για την εκτέλεση της **non-Spark** έκδοσης του αλγόριθμου **Alpha** στο **Databricks Community Edition** (1 κεντρικός κόμβος (driver) χωρίς κανένα βοηθητικό κόμβο(worker)).
- Για την εκτέλεση της **Spark** έκδοσης του αλγόριθμου **Alpha** στο **Databricks Community Edition** (1 κεντρικός κόμβος (driver) χωρίς κανένα βοηθητικό κόμβο(worker)).
- Για την εκτέλεση της **Spark** έκδοσης του αλγόριθμου **Alpha** στο **Databricks** με 1 κεντρικό κόμβο (driver) και 2 βοηθητικούς κόμβους (workers).
- Για την εκτέλεση της **Spark** έκδοσης του αλγόριθμου **Alpha** στο **Databricks** με 1 κεντρικό κόμβο (driver) και 4 βοηθητικούς κόμβους (workers).

Στις μετρήσεις αυτές φαίνεται ο συνολικός χρόνος που χρειάστηκε ο αλγόριθμος για να τρέξει, συμπεριλαμβανομένου και του χρόνου φιλτραρίσματος των δεδομένων. Σε όλους τους πίνακες ο αλγόριθμος Alpha έτρεξε για όλα το σύνολο γεγονότων με αρχικό αριθμό από ίχνη 36094. Χρησιμοποιήθηκε η τεχνική του φιλτραρίσματος που περιγράφηκε σε προηγούμενο κεφάλαιο και διατηρήθηκαν μόνο τα ίχνη εκείνα που έχουν ποσοστό εμφάνισης πάνω από την τιμή της μεταβλητής percentage. Συγκεκριμένα έγιναν οι παρακάτω μετρήσεις

Πίνακας 1

percentage = 2
readAll = true
filtering = true
Αρχικός αριθμός από ίχνη δραστηριοτήτων = 36094
Αριθμός ιχνών δραστηριοτήτων προς επεξεργασία = 7
Συνολικός αριθμός από δραστηριοτήτες = 16

	Non-Spark	**Spark (1 driver – 0 nodes)**	**Spark (1 driver – 2nodes)**	**Spark (1 driver – 4 nodes)**
Χρόνος εκτέλεσης	2.60 *λεπτά*	3.60 *λεπτά*	1.54 *λεπτά*	1.2 *λεπτά*

Για τον πίνακα 1, διατηρήθηκαν μόνο τα ίχνη δραστηριοτήτων εκείνα που έχουν ποσοστό εμφάνισης πάνω από 2%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 7

ίχνη δραστηριοτήτων καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 2%. Σε αυτήν την περίπτωση υπήρξαν 16 διαφορετικές δραστηριότητες. Παρατηρούμε ότι ο **non-Spark** αλγόριθμος Alpha είναι πιο γρήγορος από τον **Spark** αλγόριθμο Alpha όταν αυτός τρέχει σε μια συστάδα με έναν μόνο κεντρικό κόμβο (**driver**) αλλά χωρίς κανένα βοηθητικό κόμβο (**worker**). Αυτό μπορεί να δικαιολογηθεί λόγω του ότι το σύνολο δεδομένων είναι αρκετά μικρό και έτσι ο **non-Spark** αλγόριθμος Alpha μπορεί να εκτελέσει τον αλγόριθμο αρκετά γρήγορα αποφεύγοντας τον επιπλέον χρόνο που χρειάζεται ο **Spark** αλγόριθμος Alpha για να ξεκινήσει την εκτέλεση ενός Spark προγράμματος αλλά και για το serialization και deserialization των δεδομένων. Όμως ακόμα και για αυτό το μικρό σύνολο δεδομένων ο **non-Spark** αλγόριθμος Alpha είναι αρκετά πιο αργός από τον **Spark** αλγόριθμο Alpha όταν αυτός τρέχει σε μια συστάδα με έναν κεντρικό κόμβο (**driver**) και 2 ή 4 βοηθητικούς κόμβους (**workers**).

Πίνακας 2

percentage = 1
readAll = true
filtering = true
Αρχικός αριθμός από ίχνη δραστηριοτήτων = 36094
Αριθμός ιχνών δραστηριοτήτων προς επεξεργασία = 8
Συνολικός αριθμός από δραστηριοτήτες = 16

	Non-Spark	**Spark (1 driver – 0 nodes)**	**Spark (1 driver – 2nodes)**	**Spark (1 driver – 4 nodes)**
Χρόνος εκτέλεσης	2.47 *λεπτά*	3.50 *λεπτά*	1.49 *λεπτά*	*1.10 λεπτά*

Για τον πίνακα 2, διατηρήθηκαν μόνο τα ίχνη δραστηριοτήτων εκείνα που έχουν ποσοστό εμφάνισης πάνω από 1%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 8 ίχνη δραστηριοτήτων καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 1%. Σε αυτήν την περίπτωση υπήρξαν 16 διαφορετικές δραστηριότητες. Παρατηρούμε ότι ο **non-Spark** αλγόριθμος Alpha είναι πιο γρήγορος από τον **Spark** αλγόριθμο Alpha όταν αυτός τρέχει σε μια συστάδα με έναν μόνο κεντρικό κόμβο (**driver**) αλλά χωρίς κανένα βοηθητικό κόμβο (**worker**). Αυτό μπορεί να δικαιολογηθεί λόγω του ότι το σύνολο δεδομένων είναι αρκετά μικρό και έτσι ο **non-Spark** αλγόριθμος Alpha μπορεί να εκτελέσει τον αλγόριθμο αρκετά γρήγορα αποφεύγοντας τον επιπλέον χρόνο που χρειάζεται ο **Spark** αλγόριθμος Alpha για να ξεκινήσει την εκτέλεση ενός **Spark** προγράμματος αλλά και για το serialization και deserialization των δεδομένων. Όμως ακόμα και για αυτό το μικρό σύνολο δεδομένων ο **non-Spark** αλγόριθμος Alpha είναι αρκετά πιο αργός από τον **Spark** αλγόριθμο Alpha όταν αυτός τρέχει σε μια συστάδα με έναν κεντρικό κόμβο (**driver**) και 2 ή 4 βοηθητικούς κόμβους (**workers**).

Πίνακας 3

percentage = 0.8
readAll = true
filtering = true
Αρχικός αριθμός από ίχνη δραστηριοτήτων = 36094
Αριθμός ιχνών δραστηριοτήτων προς επεξεργασία = 9
Συνολικός αριθμός από δραστηριοτήτες = 17

	Non-Spark	**Spark (1 driver – 0 nodes)**	**Spark (1 driver – 2nodes)**	**Spark (1 driver – 4 nodes)**
Χρόνος εκτέλεσης	10.35 *λεπτά*	8.17 *λεπτά*	6.07 *λεπτά*	3.35 *λεπτά*

Για τον πίνακα 3, διατηρήθηκαν μόνο τα ίχνη δραστηριοτήτων εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.8%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 9 ίχνη δραστηριοτήτων καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.8%. Σε αυτήν την περίπτωση υπήρξαν 17 διαφορετικές δραστηριότητες. Παρατηρούμε ότι η **Spark** έκδοση του αλγόριθμου Alpha είναι γρηγορότερη από την **non-Spark** έκδοση του αλγόριθμου Alpha για όλες τις περιπτώσεις συστάδας υπολογιστών που χρησιμοποιήθηκαν 0,2 ή 4 κόμβοι. Αυτό οφείλεται στο γεγονός ότι το σύνολο δεδομένων είναι πιο πολύπλοκο και υπάρχουν περισσότερες δραστηριότητες που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα παίζει καταλυτικό ρόλο στην μείωση του χρόνου εκτέλεσης του αλγόριθμου Alpha. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του αλγόριθμου Alpha έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του αλγόριθμου Alpha σε μια συστάδα υπολογιστών με 4 κόμβους (10.35 λεπτά έναντι 3.35 λεπτά).

Πίνακας 4

percentage = 0.7
readAll = true
filtering = true
Αρχικός αριθμός από ίχνη δραστηριοτήτων = 36094
Αριθμός ιχνών δραστηριοτήτων προς επεξεργασία = 10
Συνολικός αριθμός από δραστηριοτήτες = 18

	Non-Spark	**Spark (1 driver – 0 nodes)**	**Spark (1 driver – 2nodes)**	**Spark (1 driver – 4 nodes)**
Χρόνος εκτέλεσης	34.60 *λεπτά*	17.35 *λεπτά*	15.40 *λεπτά*	9.92 *λεπτά*

Για τον πίνακα 4, διατηρήθηκαν μόνο τα ίχνη δραστηριοτήτων εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.7%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 10 ίχνη δραστηριοτήτων καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.7%. Σε αυτήν την περίπτωση υπήρξαν 18 διαφορετικές δραστηριότητες. Παρατηρούμε ότι η **Spark** έκδοση του αλγόριθμου Alpha είναι γρηγορότερη από την **non-**

Spark έκδοση του αλγόριθμου Alpha για όλες τις περιπτώσεις συστάδας υπολογιστών που χρησιμοποιήθηκαν 0,2 ή 4 κόμβοι. Αυτό οφείλεται στο γεγονός ότι το σύνολο δεδομένων είναι πιο πολύπλοκο και υπάρχουν περισσότερες δραστηριότητες που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα παίζει σημαντικό ρόλο στην μείωση του χρόνου εκτέλεσης του αλγόριθμου Alpha. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του αλγόριθμου Alpha έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του αλγόριθμου Alpha σε μια συστάδα υπολογιστών με 4 κόμβους (34.60 λεπτά έναντι 9.92 λεπτά).

Πίνακας 5

percentage = 0.5
readAll = true
filtering = true
Αρχικός αριθμός από ίχνη δραστηριοτήτων = 36094
Αριθμός ιχνών δραστηριοτήτων προς επεξεργασία = 12
Συνολικός αριθμός από δραστηριοτήτες = 19

	Non-Spark	**Spark (1 driver – 0 nodes)**	**Spark (1 driver – 2nodes)**	**Spark (1 driver – 4 nodes)**
Χρόνος εκτέλεσης	2.75 ***ώρες***	1.06 ***ώρες***	52.25 *λεπτά*	43.58 *λεπτά*

Για τον πίνακα 5, διατηρήθηκαν μόνο τα ίχνη δραστηριοτήτων εκείνα που έχουν ποσοστό εμφάνισης πάνω από 0.5%. Έτσι τελικά ο αλγόριθμος χρειάστηκε να επεξεργαστεί μόνο 12 ίχνη δραστηριοτήτων καθώς τα υπόλοιπα επαναλαμβάνονται ή έχουν ποσοστό εμφάνισης κάτω του 0.5%. Σε αυτήν την περίπτωση υπήρξαν 19 διαφορετικές δραστηριότητες. Παρατηρούμε ότι η **Spark** έκδοση του αλγόριθμου Alpha είναι γρηγορότερη από την **non-Spark** έκδοση του αλγόριθμου Alpha για όλες τις περιπτώσεις συστάδας υπολογιστών που χρησιμοποιήθηκαν 0,2 ή 4 κόμβοι. Αυτό οφείλεται στο γεγονός ότι το σύνολο δεδομένων είναι πιο πολύπλοκο και υπάρχουν περισσότερες δραστηριότητες που πρέπει να επεξεργαστούν και έτσι η ύπαρξη παραπάνω από ένα μηχανημάτα παίζει σημαντικό ρόλο στην μείωση του χρόνου εκτέλεσης του αλγόριθμου Alpha. Έτσι παρατηρούμε την μεγάλη διαφορά των χρόνων εκτέλεσης για την **non-Spark** έκδοση του αλγόριθμου Alpha έναντι του χρόνου εκτέλεσης για την **Spark** έκδοση του αλγόριθμου Alpha σε μια συστάδα υπολογιστών με 4 κόμβους (2.75 ώρες έναντι 43.58 λεπτά).

Από τα πειράματα που έγιναν, στους πίνακες 1 και 2 φαίνεται ότι η **non-Spark** έκδοση του αλγόριθμου Alpha είναι γρηγορότερη από την **Spark** έκδοση για μια συστάδα με κανένα κόμβο. Αυτό συμβαίνει γιατί το **Databricks Community Edition** παρέχει συστάδα μόνο με ένα κετρνικό κόμβο (driver) χωρίς κανένα βοηθητικό κόμβο (worker). Έτσι λόγω του ότι τα δεδομένα που θα επεξεργαστούν χωράνε στην κύρια μνήμη του συστήματος (RAM) και επειδή η προετοιμασία που χρειάζεται για να ξεκινήσει η εκτέλεση ενός Spark προγράμματος είναι αρκετά μεγάλη σε σχέση με το να τρέξει το ίδιο ακριβώς πρόγραμμα σε καθαρή Scala, έχουμε σαν αποτέλεσμα η εκτέλεση της **non-Spark** υλοποίησης να είναι

πιο γρήγορη. Φυσικά αυτό δεν ισχύει για την εκτέλεση της **Spark** έκδοσης για μια συστάδα με 2 ή 4 κόμβους, καθώς εκεί το Spark εκμεταλλεύεται πλήρως τις δυνατότητες της συστάδας και έτσι η εκτέλεση του αλγορίθμου είναι πολύ γρηγορότερη σε σχέση με την **non-Spark** έκδοση του αλγόριθμου Alpha.

Από τις μετρήσεις που έγιναν φάνηκε ότι το πιο χρονοβόρο σημείο του αλγόριθμου Alpha είναι η εύρεση των causal groups. Η εύρεση των causal groups είναι μια πολύ βαριά διαδικασία γιατί το πρόγραμμα πρέπει να υπολογίσει πολλούς συνδιασμούς από σύνολα και υποσύνολα από δραστηριότητες ενώ ταυτόχρονα πρέπει να ελέγχει αν τα ζεύγη μεταξύ των δραστηριότητων ανηκούν σε συγκεκριμένες σχέσεις. Σε αυτό το στάδιο παρατηρούμε ότι η **Spark** υλοποίηση είναι πολύ πιο γρήγορη είτε τρέχει σε συστάδα χωρίς κανένα κόμβο είτε σε συστάδα με πολλούς κόμβους.

7. Συμπεράσματα και Προοπτικές

Στο κεφάλαιο αυτό γίνεται μια παρουσίαση των συμπερασμάτων που έγιναν κατά την διάρκεια της εκπόνησης της παρούσας διπλωματικής εργασίας αλλά και τις προοπτικές επέκτασεις που έχει το θέμα που αναλύθηκε.

7.1 Συμπεράσματα

Σύμφωνα με τις μετρήσεις που παρουσιάστηκαν στο έκτο κεφάλαιο παρατηρούμε ότι υπάρχει σημαντική μείωση του συνολικού χρόνου εκτέλεσης του αλγόριθμου Alpha όταν αυτός παραλληλοποιείται με την χρήση του περιβάλλοντος ανάπτυξης εφαρμογών Spark. Αυτό οφείλεται στο γεγονός ότι η επεξεργασία των δεδομένων γίνεται σε παραπάνω από έναν κόμβους σε μια συστάδα υπολογιστών με την χρήση του Spark το οποίο κατανέμει τα δεδομένα στους κόμβους αυτούς. Αν συνυπολογίσουμε το γεγονός ότι η υλοποίηση της Spark έκδοσης του αλγορίθμου δεν διαφέρει αρκετά σε σχέση με την non-Spark έκδοση καθώς χρησιμοποιούνται παρόμοιοι τελεστές και στις δύο εκδόσεις γίνεται αντιληπτό ότι η υλοποίηση με την χρήση του Spark μπορεί να δώσει πάρα πολλά πλεονεκτήματα σε κάποιον προγραμματιστή που επιθυμεί να εξάγει ένα γράφημα με τις διαδικασίες που συμβαίνουν μέσα σε μια επιχείρηση χωρίς να χρειαστεί να μάθει ένα περιβάλλον υλοποίησης το οποίο διαφέρει αρκετά σε σχέση με τις γνώσεις που ήδη έχει. Ακόμα επιβεβαιώθηκε το γεγονός ότι για προγράμματα που δεν απαιτούν πολλούς υπολογισμούς και δεν χρησιμοποιούν μέγαλο σύνολο δεδομένων σαν είσοδο είναι καλύτερο να χρησιμοποιείται ένα μηχάνημα για την επεξεργασία των δεδομένων και όχι κάποια συστάδα υπολογιστών ώστε να αποφεύγεται ο επιπλέον χρόνος που χρειάζεται το **Spark** για να ξεκινήσει την εκτέλεση ενός **Spark** προγράμματος αλλά και για το serialization και deserialization των δεδομένων.

7.2 Προοπτικές

Στα πλαίσια αυτής της διπλωματικής εργασίας υλοποιήθηκε ο αλγόριθμος εξόρυξης διαδικασιών Alpha σε περιβάλλον ανάπτυξης Spark. Θα είχε αρκετό ενδιαφέρον η μελέτη του Spark Streaming ώστε να μελετηθεί η συμπεριφόρα του υλοποιημένου παράλληλου αλγορίθμου όταν εμφανίζονται νέα γεγόνοτα σε πραγματικό χρόνο. Ακόμα θα μπορούσε να γίνει υλοποίηση και άλλων αλγόριθμων εξόρυξης διαδικασιών σε Spark όπως ο αλγόριθμος Flexible Heuristic Miner και να γίνει σύγκριση των χρόνων εκτέλεσης σε σχέση με αυτούς του αλγόριθμου Alpha. Τέλος θα μπορούσε να γίνει υλοποίηση του αλγόριθμου Alpha στο περιβάλλον ανάπτυξης Apache Flink το οποίο είναι ένα περιβάλλον για

κατανεμημένη εκτέλεση υπολογισμών σε ένα σύνολο δεδομένων και να συγκριθούν οι χρόνοι εκτέλεσης με αυτούς του αλγόριθμου Alpha.

[1] Alpha Algorithm, http://mlwiki.org/index.php/Alpha_Algorithm

[2] Apache Spark Architecture, https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338

[3] Apache Spark Caching, https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd-caching.html

[4] Apache Spark Datasets, https://databricks.com/spark/getting-started-with-apache-spark/datasets

[5] Apache Spark Driver, https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-driver.html

[6] Apache Spark Executor, https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-Executor.html

[7] Apache Spark RDDs, https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd.html

[8] Apache Spark RDDs, https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

[9] Apache Spark Tuning, https://spark.apache.org/docs/latest/tuning.html

[10] Cluster overview, https://spark.apache.org/docs/latest/cluster-overview.html

[11] Databricks Community Edition, https://community.cloud.databricks.com

[12] Frank Kane, “Apache Spark 2 with Scala – Hands on with Big Data”, www.udemy.com

[13] Java Garbage collection for Spark applications, https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

[14] Jianmin Wang, Raymond K. Wong, Jianwei Ding, Qinlong Guo and Lijie Wen, "Efficient Selection of Process Mining Algorithms," in IEEE Transactions on Services Computing.

[15] Joerg Evermann (2016), “Scalable Process Discovery Using Map-Reduce”, IEEE Transactions on Services Computing

[16] Petar Zecevic, Marko Bonaci, “Spark in Action”, Manning Publications CO

[17] Petri Nets, http://mlwiki.org/index.php/Petri_Nets

[18] Scala Programming Language, https://en.wikipedia.org/wiki/Scala_(programming_language)

[19] SQL programming guide, https://spark.apache.org/docs/latest/sql-programming-guide.html

[20] Wil van der Aalst, “Process Mining: Data science in Action”, www.coursera.org

[21] Wil van der Aalst, A.J.M.M. (Ton) Weijters, Laura Măruşter. (2004). Workflow Mining: Discovering Process Models from Event Logs, IEEE Transactions on Knowledge and Data Engineering.

[22] Wil van der Aalst (2013). Decomposing Petri nets for process mining : a generic approach. Distributed and Parallel Databases.