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

Μελέτη επίδοσης συστημάτων επεξεργασίας ροών δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΣΚΟΥΡΑ ΕΛΕΝΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Αθήνα, Ιούλιος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Μελέτη επίδοσης συστημάτων επεξεργασίας ροών δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΣΚΟΥΡΑ ΕΛΕΝΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Ιουλίου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
……...	……..	……..
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Δημήτριος Τσουμάκος
Καθηγητής Ε.Μ.Π	Επίκουρος Καθηγητής Ε.Μ.Π	Αναπληρωτής Καθηγητής Ιονίου Πανεπιστημίου

Αθήνα, Ιούλιος 2020

(Υπογραφή)

……...

ΣΚΟΥΡΑ ΕΛΕΝΗ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη για την επίβλεψη αυτής της διπλωματικής εργασίας καθώς και για την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο Υπολογιστικών Συστημάτων. Επίσης ευχαριστώ ιδιαίτερα την Δρ Κατερίνα Δόκα για την καθοδήγηση της και την εξαιρετική συνεργασία που είχαμε.Τέλος θα ήθελα να ευχαριστήσω τους γονείς μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Περίληψη

Τα τελευταία χρόνια παρατηρείται ένα αυξανόμενο ενδιαφέρον σε σχέση με τον τομέα των big data στην πληροφορική. Λόγω και της ανάπτυξης του Internet of things και των social media , τα τελευταία χρόνια παράγεται ένας μεγάλος όγκος από streaming δεδομένα με ένα πρωτοφανή ρυθμό και αυτός ο όγκος πρέπει να τεθεί υπό διαχείριση με ένα χρονικά ορισμένο τρόπο. Για αυτό το λόγο καθίσταται αναγκαία η επεξεργασία big data δεδομένων σε **real-time χρόνο** .Η ανάγκη για scalable και αποτελεσματική stream analysis έχει οδηγήσει στην ανάπτυξη πολλών open-source streaming data processing systems (SDPSs) με ιδιαίτερα αποκλίνουσες δυνατότητες και χαρακτηριστικά σε σχέση με την επίδοση.

Σκοπός της διπλωματικής εργασίας είναι η διεξοδική μελέτη του performance δύο ευρέως σε χρήση SDPSs , ειδικότερα του Apache Spark και του Apache Flink. Η μελέτη μας εστιάζει κατά βάση στον υπολογισμό του throughput και του latency windowed operations, που είναι και ο βασικός τύπος operations που χρησιμοποιείται στα stream analytics. Για αυτό το benchmark, παράγουμε streaming big data δεδομένα τα οποία ανάλογα με το use-case μεταβάλλονται σε σχέση με ένα πλήθος παραμέτρων**.**

Αυτή η μελέτη μπορεί να βοηθήσει το χρήστη στην επιλογή του κατάλληλου engine για την επεξεργασία streaming big data δεδομένων ανάλογα με το είδος της εφαρμογής , τον τύπο των δεδομένων και των resources που διαθέτει.

Λέξεις Κλειδιά

Big data, SDPSs, streaming data, real-time,Apache Spark , Apache Flink , benchmark, throughput, latency

Abstract

Over the last few years Big data technology has gained an increasing interest . As a result of the IOT and social media development, a huge volume of streaming data is generated at an unprecedented rate. This volume of streaming data needs to be handled in a timely fashion. Thus, big data processing needs to take place in real-time .The need for scalable and efficient stream analysis has led to the development of many open-source streaming data processing systems (SDPSs) with highly diverging capabilities and performance characteristics.
The aim of this thesis is to evaluate the performance of two widely used SDPSs in detail, namely Apache Spark, and Apache Flink. Our evaluation focuses in particular on measuring the throughput and latency of windowed operations, which are the basic type of operations in stream analytics. For this benchmark, we produce streaming big data , which vary depending on the use-cases**.**
This thesis can help the users to choose the appropriate processing engine depending on the application , the data type and the available resources.

Keywords

Big data, SDPSs, streaming data,real-time, Apache Spark , Apache Flink , benchmark, throughput, latency

Πίνακας περιεχομένων

ΚΕΦΑΛΑΙΟ 1: Εισαγωγή

1.1 Big Data

Τα τελευταία χρόνια παρατηρείται ένα αυξανόμενο ενδιαφέρον σε σχέση με τον τομέα των big data στην πληροφορική. Τα big data είναι ένας τομέας που χειρίζεται τρόπους για να αναλύει, να εξάγει συστηματικά πληροφορίες από ή αλλιώς να διαχειρίζεται datasets τα οποία είναι τόσο μεγάλα και πολύπλοκα για να μπορούν να τεθούν υπό διαχείριση από παραδοσιακό data-processing application software .
Λόγω και της ανάπτυξης του Internet of things, παράγεται ένας μεγάλος όγκος από streaming data με ένα πρωτοφανή ρυθμό και αυτός ο όγκος πρέπει να τεθεί υπό διαχείριση με ένα χρονικά ορισμένο τρόπο. Για αυτό το λόγο καθίσταται αναγκαία η επεξεργασία big data δεδομένων σε real-time χρόνο .
Το **stream processing** είναι μια big data τεχνολογία που παρέχει τη δυνατότητα επεξεργασίας streaming data σε real-time χρόνο . Επεξεργάζεται data streams εντός ενός μικρού χρονικού διαστήματος από την στιγμή που έχει λάβει τα data. Ο χρόνος επεξεργασίας ποικίλλει από ms σε λεπτά.
Υπάρχουν πολλά processing engines για την διαχείριση streaming big data δεδομένων που απαντούν στην ανάγκη για scalable και αποτελεσματική stream analysis. Παρά την διαθεσιμότητα αυτή , είναι δύσκολο για τους χρήστες να γνωρίζουν ποιο processing engine είναι καλύτερο ανάλογα με τις ανάγκες τους .

1.2 Αντικείμενο διπλωματικης

Σκοπός της διπλωματικής εργασίας είναι να βοηθήσει το χρήστη στην επιλογή του κατάλληλου engine για την επεξεργασία streaming big data δεδομένων ανάλογα με το είδος της εφαρμογής , τον τύπο των δεδομένων και των resources που διαθέτει. Συγκεκριμένα επιλέγοντας τα πιο δημοφιλή engines , το **Spark** και το **Flink**, διεξάγουμε μια μελέτη στην οποία συγκρίνουμε το performance των δύο engines σε σχέση με το τύπο της εφαρμογής , τον τύπο των δεδομένων και των διαθέσιμων resources . Επειδή η μελέτη αφορά streaming δεδομένα , χρησιμοποιούμε ένα extension του Spark, το Spark Streaming. Η μελέτη μας εστιάζει κατά βάση στον υπολογισμό του throughput και του latency windowed operations, που είναι και ο βασικός τύπος operations που χρησιμοποιείται στα stream analytics.

Στο 3ο και 4ο κεφάλαιο περιγράφουμε διεξοδικά τα δύο engines αναφορικά με τη αρχιτεκτονική τους, τον τρόπο λειτουργίας τους και τα χαρακτηριστικά τους. Επίσης στο 5ο κεφάλαιο περιγράφεται διεξοδικά ο τρόπος λειτουργίας του Kafka καθώς μέσω του Kafka διοχετεύουμε τα big data δεδομένα στα δύο engines για την διεξαγωγή όλων των πειραμάτων.

ΚΕΦΑΛΑΙΟ 2: Θεωρητικό υπόβαθρο

2.1 Τι είναι το stream processing

Το Stream processing είναι μια big data τεχνολογία.Συγκεκριμένα είναι η επεξεργασία δεδομένων τη στιγμή την οποία τα δεδομένα δημιουργούνται. Η πλειονότητα των data δημιουργούνται ως continuous streams: sensor events , δραστηριότητα χρηστών σε ένα website ,οικονομικές συναλλαγές κ.α. Ένα stream είναι μια ατέρμονη ακολουθία από (tuple, timestamp) pairs.Τα data δημιουργούνται προοδευτικά ως μια ακολουθία από events (Το event είναι το ίδιο με το (tuple,timestamp) pair).
Πριν από το stream processing , αυτά τα δεδομένα συχνά αποθηκεύονταν σε databases, filesystems ή άλλες μορφές μαζικής αποθήκευσης. Τα διάφορα applications έθεταν queries στα data και έκαναν υπολογισμούς πάνω στα data αφότου είχαν αποθηκευτεί σε κάποιο storage system όπως φαίνεται και στην παρακάτω εικόνα.

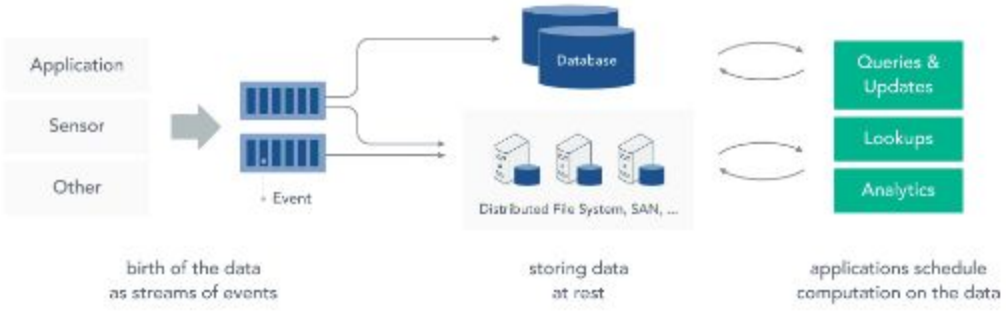


Fig. 1: Before stream processing: a data-at-rest infrastructure

Το stream processing ανατρέπει αυτό το πρότυπο. Η λογική , τα analytics και τα queries (transformations) των applications υφίστανται αδιάκοπα, και τα data ρέουν διαμέσου αυτών αδιάλειπτα.

Κατά τη λήψη ενός event από ένα stream, ένα stream processing application αντιδρά σε αυτό το event: η λήψη του event μπορεί να πυροδοτήσει ένα action , update ενός aggregate η άλλου statistic , ή απλώς την αποθήκευση αυτού του event για μελλοντική χρήση .
Οι streaming υπολογισμοί μπορούν να επεξεργαστούν πολλαπλά data streams από κοινού , και κάθε υπολογισμός πάνω σε ένα event data stream να δημιουργήσει άλλα event data streams.

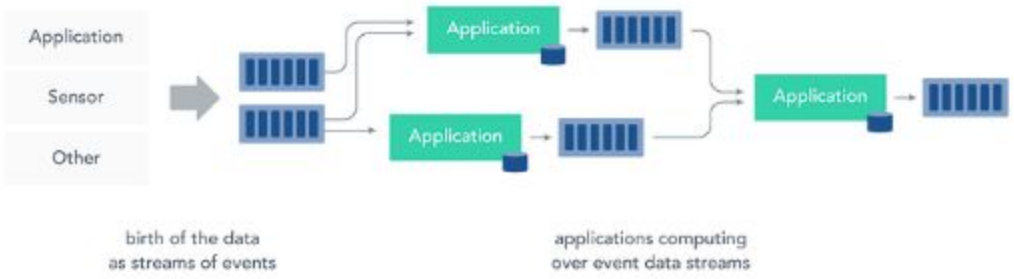


Fig. 2: A stream processing infrastructure

Τα συστήματα τα οποία λαμβάνουν και αποστέλλουν data streams και εκτελούν τα applications η analytics logic ονομάζονται stream processors. Οι βασικές αρμοδιότητες ενός stream processor είναι να διασφαλίσει ότι τα data ρέουν αποτελεσματικά και το computation κλιμακώνει και είναι fault-tolerant.

Για παράδειγμα το **Flink** είναι ισχυρό open source stream processing framework το οποίο ανταποκρίνεται σε αυτές τις προκλήσεις.

Το πρότυπο του stream processing αντιμετωπίζει πολλά challenges που οι developers των real-time data analytics και event-driven applications αντιμετωπίζουν σήμερα. Κάποια από τα challenges που αντιμετωπίζει είναι:

1) Applications και analytics αντιδρούν στα events αμέσως: Δεν υπάρχει καθυστέρηση μεταξύ " συμβαίνει ένα event "-> " εξαγωγή πληροφορίας" -> " λήψη action" . Τα actions και τα analytics είναι up-to-date , αντανακλώντας τα data ενώ είναι ακόμη φρέσκα , ουσιαστικά και πολύτιμα.
2)Το stream processing μπορεί να χειριστεί μεγέθη data που είναι πολύ μεγαλύτερα από άλλα data processing systems: Τα event streams φορτώνονται αμέσως , και μόνο ένα ουσιαστικό υποσύνολο των data αποθηκεύεται.
3) Το stream processing εύκολα και φυσικά μοντελοποιεί τη αδιάκοπη και χρονικά ορισμένη φύση των data : Αυτό έρχεται σε αντίθεση με προγραμματισμένα (batch) queries και

analytics πάνω σε στατικά δεδομένα. Η υλοποίηση σταδιακών updates , αντί για τον περιοδικό επανυπολογισμό όλων των data ,εφαρμόζεται φυσικά με το stream processing μοντέλο.
4)Το stream processing αποκεντρώνει και χωρίζει τις δομές αποθήκευσης: Το streaming πρότυπο μειώνει την ανάγκη για μεγάλες και ακριβές κατανεμημένες databases. Αντ' αυτού, κάθε processing application διατηρεί το δικό του state και δεδομένα, τα οποία κατασκευάζονται απλά από το stream processing framework. Με αυτόν τον τρόπο , ένα stream processing application χωράει φυσικά σε microservices αρχιτεκτονικές.

2.2 Stateful Stream Processing

Το stateful stream processing είναι ένα υποσύνολο του stream processing όπου ένα computation διατηρεί contextual **state**. Το state χρησιμοποιείται για την αποθήκευση πληροφοριών που προέρχονται από previously-seen events.

Ουσιαστικά όλα τα σημαντικά stream processing applications χρειάζονται stateful stream processing. Για παράδειγμα:

1) Ένα application που αποτρέπει απάτες θα διατηρήσει τις τελευταίες συναλλαγές για κάθε πιστωτική κάρτα στο state της. Κάθε καινούρια συναλλαγή με αυτές που υπάρχουν στο state του application , χαρακτηρίζεται ως valid ή ύποπτη απάτης, και το state ανανεώνεται με αυτό το transaction.
2) Ένα online συμβουλευτικό application θα κρατήσει παραμέτρους που περιγράφουν τις προτιμήσεις του χρήστη. Κάθε αλληλεπίδραση του χρήστη με ένα προϊόν δημιουργεί ένα event που ανανεώνει αυτές τις παραμέτρους.
3) Ένα microservice το οποίο χειρίζεται ένα song playlist ή e-commerce shopping cart λαμβάνει events για κάθε αλληλεπίδραση του χρήστη με τραγούδια ή προϊόντα . Το state διατηρεί λίστα με όλα τα αντικείμενα που έχουν προστεθεί.

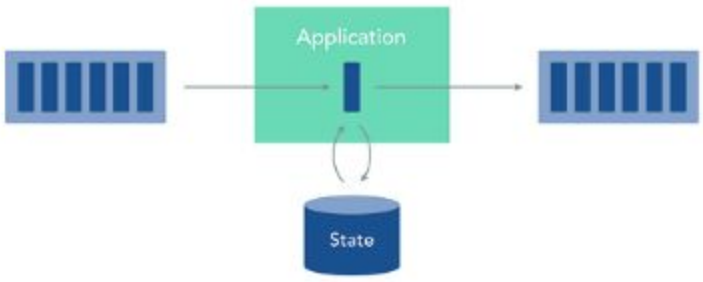


Fig. 3: Stateful stream processing

Εννοιολογικά , το stateful stream processing φέρνει μαζί το database (ή key/value store tables) και το event-driven /reactive application (ή analytics logic) σε μια στενά-ενσωματωμένη οντότητα.
Η βαθιά ενσωμάτωση μεταξύ του state και της εκτέλεσης ενός application οδηγεί σε υψηλό performance , κλιμακωσιμότητα , data consistency και operational simplicity.
Το stateful stream processing απαιτεί ένα stream processor που να υποστηρίζει το state management. Το **Flink** υποστηρίζει το stateful stream processing.

2.3 Micro-batch processing

Micro-batch processing είναι η εφαρμογή συλλογής δεδομένων σε μικρά groups (batches) με σκοπό την επεξεργασία αυτών των **streaming** δεδομένων. Σε αντίθεση με το παραδοσιακό **batch processing,** το οποίο συχνά υποδηλώνει την επεξεργασία σε ένα μεγαλύτερο group από δεδομένα. Το micro-batch processing είναι μια παραλλαγή του batch processing καθώς η επεξεργασία των data συμβαίνει πιο συχνά έτσι ώστε μικρότερα groups από καινούρια data να υφίστανται επεξεργασία.
Στο batch processing και στο micro-batch processing, τα δεδομένα συλλέγονται με βάση ένα προκαθορισμένο κατώφλι ή συχνότητα προτού ξεκινήσει το processing. Ένα micro-batch μπορεί να φορτώνει data βασιζόμενο σε κάποια συχνότητα , για παράδειγμα , μπορούν να φορτώνονται δεδομένα κάθε δύο λεπτά η δύο second . Επίσης ένα micro-batch μπορεί να φορτώνει data βασιζόμενο σε κάποιο event flag ή trigger (τα δεδομένα να είναι μεγαλύτερα από δύο megabytes για παράδειγμα.

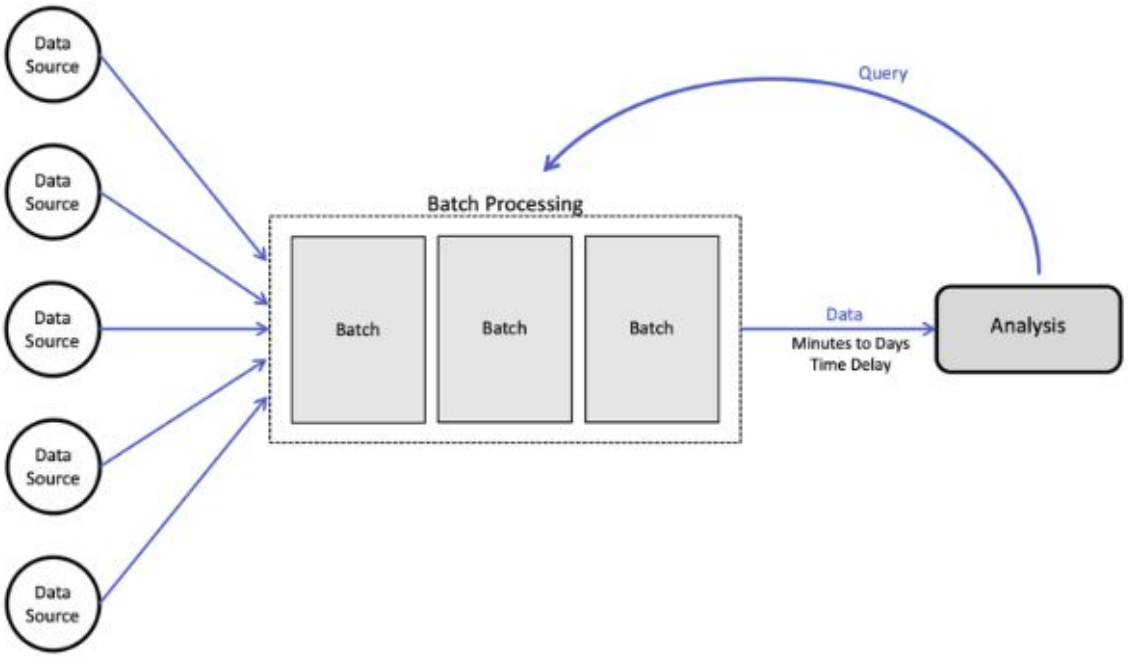


Fig. 4: Batch processing paradigm

Για παράδειγμα το **Spark streaming** είναι ένα ισχυρό open source framework το οποίο υποστηρίζει batch και micro-batch processing και είναι μια δημοφιλής επιλογή για streaming big data processing. Επίσης το Spark Streaming υποστηρίζει **stateful** micro-batch processing.

Συγκεκριμένα στο Spark Streaming τα εισερχόμενο stream δεδομένων χωρίζεται σε batches , . Τα batches πηγαίνουν για επεξεργασία κάνοντας χρήση operations (όπως map , reduce, join κ.ο.κ). Τα αποτελέσματα αυτών των operations επιστρέφονται σε batches. Ως εκ τούτου το Spark Streaming δεν εκτελεί real-time processing αλλά το προσεγγίζει.Το micro-batch processing αποτελεί έναν από τους περιορισμούς του Spark streaming

2.4 Διαφορές μεταξύ Batch και Stream Processing

Παρόλο που φαίνεται ότι οι διαφορές μεταξύ stream και batch processing , ειδικότερα micro-batch είναι μόνο ένα ζήτημα μιας μικρής διαφοράς στο timing, στην πραγματικότητα υπάρχουν διαφορές σε σχέση με την αρχιτεκτονική των δύο data processing systems και με τα applications που τα χρησιμοποιούν
Συστήματα για stream processing είναι σχεδιασμένα να ανταποκρίνονται στα data όταν αυτά έρχονται . Αυτό απαιτεί από τα συστήματα αυτά να εφαρμόζουν event-driven αρχιτεκτονική , μια αρχιτεκτονική στην οποία ο εσωτερικός workflow του συστήματος είναι σχεδιασμένος για να

παρακολουθεί συνεχώς την έλευση καινούριων data και να ξεκινάει το processing αμέσως μόλις έρθουν τα data. Από την άλλη , ο εσωτερικός workflow σε ένα batch processing system ελέγχει για καινούρια data περιοδικά και φορτώνει τα data μόνο όταν ξεκινάει το επόμενο batch interval. Η διαφορά μεταξύ stream και batch processing είναι επίσης σημαντική για τα applications. Τα applications φτιαγμένα για batch processing από τον ορισμό τους φορτώνουν data με μια καθυστέρηση .Σε ένα data pipeline με πολλαπλά βήματα , αυτές οι καθυστερήσεις συσσωρεύονται .
Επιπρόσθετα στο **micro-batch processing** ,το lag μεταξύ της έλευσης των καινούριων data και της επεξεργασίας αυτών των data ποικίλλει ανάλογα με το χρόνο μέχρι το επόμενο batch processing window-- σε καθόλου χρόνο σε κάποιες περιπτώσεις μέχρι και τον πλήρη χρόνο που μεσολαβεί μεταξύ δύο διαδοχικών batches ,για data που φτάνουν αμέσως μετά την έναρξη του processing του πρώτου batch (αυτά τα data ανήκουν στο δεύτερο batch, άρα για να φορτωθούν στο δεύτερο batch θα πρέπει να περιμένουν να τελειώσει την processing το πρώτο batch).Συνεπώς , τα batch processing applications (και οι χρήστες τους) δεν μπορούν να βασιστούν σε consistent response time και χρειάζεται να προσαρμόζονται σύμφωνα με αυτό το inconsistency και σε μεγαλύτερο latency.

Το micro-batch processing δεν είναι real-time processing, ωστόσο υποστηρίζει κάποια real-time use cases στις οποίες τα data δεν χρειάζεται να είναι up-to-the-moment ακριβή. Για παράδειγμα, μπορούμε να έχουμε corporate dashboards που ανανεώνονται κάθε 15 λεπτά ή κάθε μια ώρα.’Η μπορεί να συλλέγουμε server logs σε μικρά regular intervals για ιστορικό record-keeping

Το stream processing από την άλλη είναι απαραίτητο για use-cases που απαιτούν live αλληλεπίδραση και real-time απόκριση. Η διεξαγωγή οικονομικών συναλλαγών , η real-time fraud ανίχνευση και το real-time pricing είναι παραδείγματα στα οποία ταιριάζει το stream processing.

ΚΕΦΑΛΑΙΟ 3: Spark

3.1 Εισαγωγή

Το βασικό **abstraction** του Spark είναι το Resilient Distributed Dataset (Rdd). Τα Rdds αποτελούν τα building blocks κάθε application του Spark . Είναι fault tolerant κατανεμημένα datasets όπως υποδηλώνει και το πλήρες όνομα τους (Resilient Distributed Dataset).

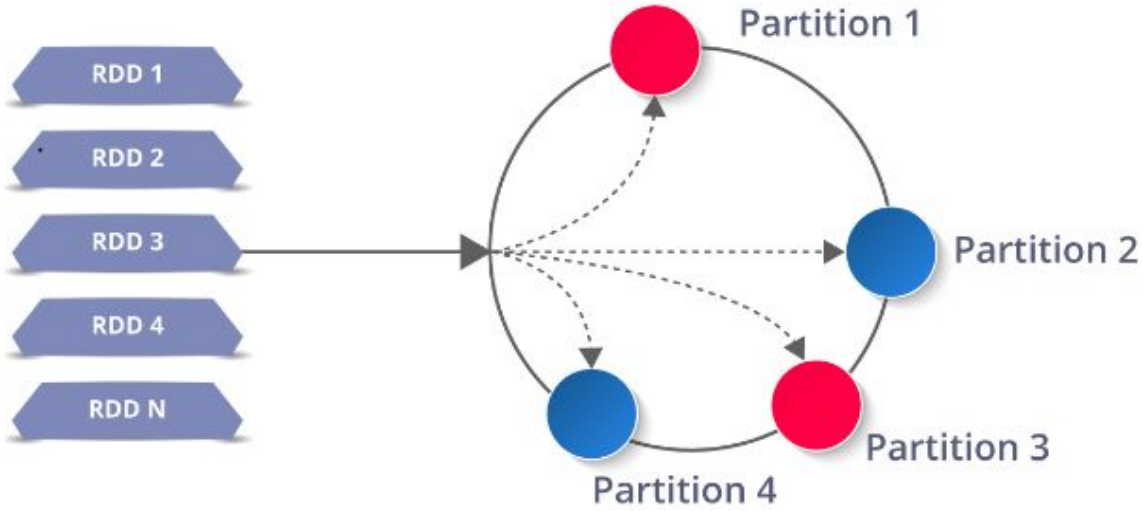


Fig. 5: RDD

Το Rdd είναι από τη φύση του αδιάστατο και ακολουθεί lazy transformations(Το Spark καθυστερεί την εκτέλεση των operations πάνω σε ένα Rdd μέχρι να είναι απολύτως απαραίτητο) Το Rdd είναι χωρισμένο σε chunks. Τα Rdds είναι πολύ ανθεκτικά , μπορούν να ανακτηθούν γρήγορα από κάθε σφάλμα καθώς τα chunks έχουν αντίγραφα σε πολλούς executor nodes. Έτσι , ακόμα και ένας node να τεθεί εκτός λειτουργίας λόγω σφάλματος , κάποιος άλλος node θα αναλάβει την επεξεργασία των δεδομένων.

Το δεύτερο βασικό **abstraction** του Spark είναι το directed acyclic graph (DAG).

Κάθε φορά που ένα application υποβάλλεται στο Spark , ο driver μετατρέπει το user κώδικα που περιέχει transformations και actions σε ένα προσανατολισμένο ακυκλικό γράφο που ονομάζεται DAG. Ο DAG είναι ένας γράφος ο οποίος καταγράφει τα operations τα οποία εφαρμόζονται σε ένα Rdd. Συγκεκριμένα είναι ένα σύνολο από κορυφές και πλευρές , όπου οι κορυφές αντιπροσωπεύουν τα Rdds και οι πλευρές αντιπροσωπεύουν το operation που πρέπει να εφαρμοστεί σε ένα Rdd. Στον DAG του Spark κάθε πλευρά κατευθύνεται από τη μια κορυφή στην άλλη .Το DAG περιλαμβάνει μια ακολουθία από κορυφές έτσι ώστε η κάθε πλευρά να κατευθύνεται από την πιο πρώιμη(πλευρά) στην μεταγενέστερη της ακολουθίας

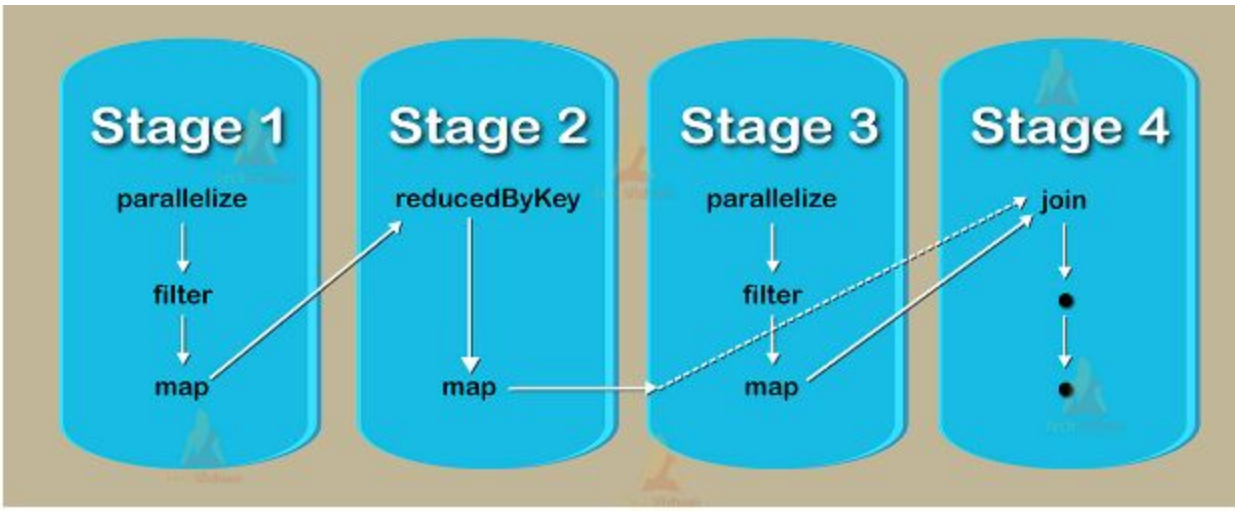


Fig. 6: DAG Visualization

(Η κορυφή είναι εκεί που είναι το parallelize που εφαρμόζεται στο Rdd και το edge είναι το sequence του parallelize-filter-map)

3.2 Αρχιτεκτονική του Spark

Στην παρακάτω εικόνα φαίνεται η βασική αρχιτεκτονική του Spark .

Στον **master node** ,έχουμε το driver program το οποίο οδηγεί το εκάστοτε application. Εσωτερικά του driver program το πρώτο πράγμα που γίνεται είναι να δημιουργηθεί το Sparkcontext. Το Sparkcontext στην ουσία είναι η είσοδος για όλες τις λειτουργίες του Spark.

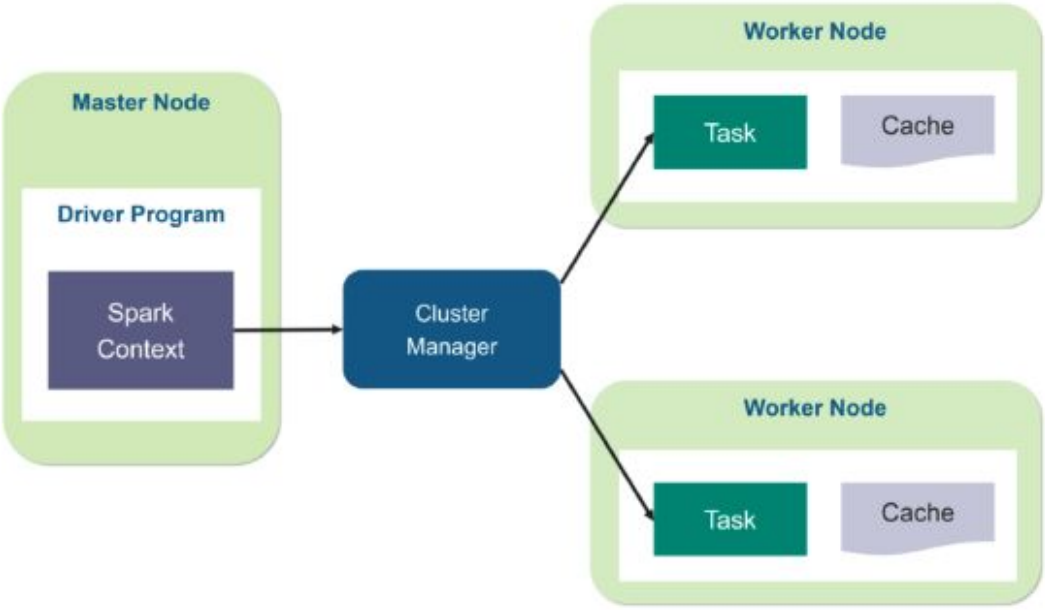


Fig. 7: Spark Architecture

Το Sparkcontext δουλεύει μαζί με τον cluster manager για να διαχειριστούν διάφορες εργασίες. Το driver program και το Sparkcontext φροντίζουν για το job execution μέσα στον cluster. Κάθε job χωρίζεται σε πολλαπλά tasks τα οποία κατανέμονται στους worker nodes του cluster. Κάθε φορά που δημιουργείται ένα Rdd μπορεί να κατανεμηθεί σε πολλαπλούς worker nodes και να γίνει cached σε κάθε node.
Οι **worker nodes** είναι οι slave nodes των οποίων η λειτουργία είναι να εκτελούν τα tasks. Αυτά τα tasks εκτελούνται πάνω στα κατανεμημένα Rdds σε κάθε worker node και στη συνέχεια επιστρέφεται το αποτέλεσμα του κάθε task στο Sparkcontext.
Ανακεφαλαιώνοντας, το Sparkcontext παίρνει το εκάστοτε job, το χωρίζει σε tasks τα οποία τα κατανέμει στους worker nodes. Τα tasks εκτελούνται πάνω στο κατανεμημένο Rdd, εκτελούν operation και στη συνέχεια συλλέγουν τα αποτελέσματα και τα επιστρέφουν πίσω στο Sparkcontext.

3.3 Job execution in Spark

Η παρακάτω εικόνα δείχνει πως λειτουργεί ο Dag κατά τη διάρκεια της εκτέλεσης ενός job στο Spark.

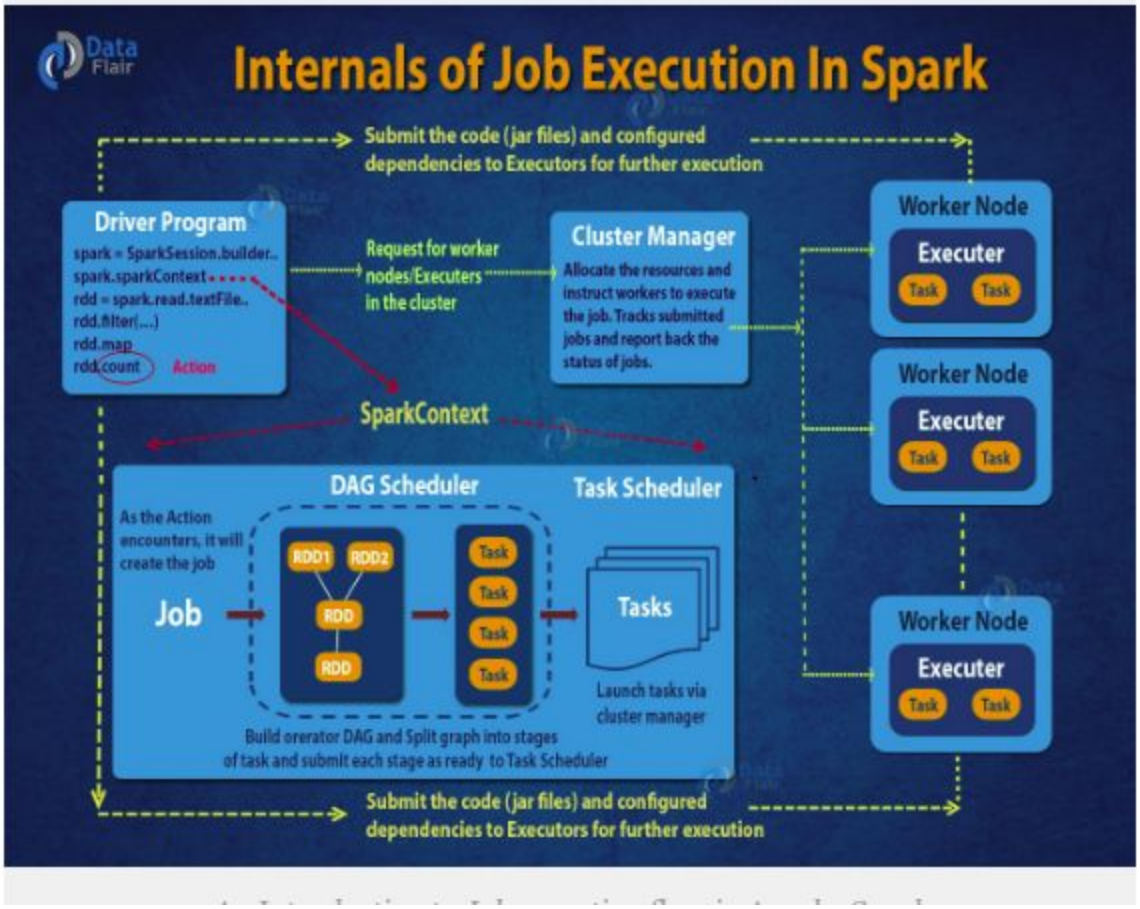


Fig. 7: An introduction to Job execution flow in Apache Spark

Κάθε φορά που υποβάλλεται ένα application code στο Spark , ο driver μετατρέπει το application code που περιλαμβάνει transformations και actions σε ένα κατευθυνόμενο ακυκλικό γράφο (DAG). Όταν καλείται ένα action, το Spark υποβάλλει το Dag στον Dag scheduler. Ο Dag scheduler μετατρέπει το Dag (logical graph) σε ένα φυσικό πλάνο εκτέλεσης με πολλά stages. Αφότου έχει δημιουργηθεί το φυσικό πλάνο εκτέλεσης , ο Dag scheduler δημιουργεί physical μονάδες εκτέλεσης για κάθε stage που ονομάζονται tasks. Στη συνέχεια ο Dag scheduler αναθέτει τα tasks στον Task Scheduler .Task scheduler ουσιαστικά σημαίνει υποβολή tasks για εκτέλεση. Ο task scheduler δεν γνωρίζει για τα dependencies των tasks. Για αυτό το λόγο ο Dag scheduler είναι υπεύθυνος για να αποφασίσει τα προτιμώμενα locations για να τρέξει το κάθε task αντίστοιχα. Τα tasks ξεκινούν την εκτέλεση τους διαμέσου του cluster manager. Για να ξεκινήσει η εκτέλεση των tasks θα πρέπει ο driver να διαπραγματευτεί με τον cluster manager για τα resources. Ο cluster manager ξεκινάει τους executors στους worker nodes για λογαριασμό του driver. Σε αυτό το σημείο , ο driver θα στείλει τα tasks στους executors βασιζόμενος στο data placement. Ο driver έχει μια πλήρη εικόνα των executors που εκτελούν τα

tasks. Κατά τη διάρκεια της εκτέλεσης των tasks , το driver program παρακολουθεί το set των executors που τρέχουν. Ο driver επίσης προγραμματίζει την εκτέλεση μελλοντικών tasks βασιζόμενος στο data placement.

3.4 Fault tolerance in Spark

Όπως ξέρουμε ο DAG κρατάει record των operations που εφαρμόζονται σε ένα RDD. Κρατάει οποιαδήποτε λεπτομέρεια των tasks που εκτελούνται σε διαφορετικά partitions του Spark Rdd. Έτσι τη στιγμή που συμβαίνει ένα σφάλμα ή ακόμα και στην απώλεια ενός Rdd , μπορούμε να το επαναφέρουμε με τη βοήθεια του DAG graph. Για παράδειγμα ένα operation εκτελείται και κάποια στιγμή ένα Rdd καταστρέφεται. Με τη βοήθεια του cluster manager , αναγνωρίζεται η partition στην οποία συνέβη η απώλεια . Μετά από αυτό διαμέσου του DAG ανατίθεται το partition που καταστράφηκε σε έναν καινούριο node. Με αυτόν τρόπο επαναφέρεται το Rdd στην κατάσταση πριν από το σφάλμα και συνεχίζεται η εκτέλεση .

3.5 Spark Streaming

Αποτελεί ένα extension του Spark API το οποίο επιτρέπει την επεξεργασία streaming δεδομένων. Τα δεδομένα μπορούν να προσληφθούν από διάφορα sources όπως ο Kafka, Flume, Kinesis ή TCP sockets και μπορούν να τεθούν σε επεξεργασία κάνοντας χρήση high-level συναρτήσεων. Τα δεδομένα που έχουν υποστεί επεξεργασία μπορούν να αποθηκευτούν σε filesystems , βάσεις δεδομένων και dashboards όπως φαίνεται και στην παρακάτω εικόνα.

Fig. 8: Spark Streaming architecture

Εσωτερικά λειτουργεί ως εξής . Το Spark streaming λαμβάνει τα streaming δεδομένα και τα χωρίζει σε batches ενός προκαθορισμένου interval. Το κάθε batch αντιμετωπίζεται ως ένα Rdd.

Στη συνέχεια αυτά τα Rdds υφίστανται επεξεργασία από το Spark engine για να παραχθεί το τελικό stream των αποτελεσμάτων σε batches.

Fig. 9: Spark Streaming data flow

Το Spark streaming διαθέτει ένα high-level abstraction που ονομάζεται discretized stream η Dstream , το οποίο αναπαριστά ένα συνεχές stream δεδομένων. Τα Dstreams μπορούν να δημιουργηθούν από εισερχόμενα streams δεδομένων από πηγές όπως ο Kafka, Flume, Kinesis η εφαρμόζοντας high-level operations σε άλλα Dstreams. Εσωτερικά , ένα Dstream αναπαρίσταται ως μια ακολουθία από Rdds. Το Rdd είναι το κύριο abstraction που παρέχει το Spark το οποίο είναι μια συλλογή δεδομένων κατανεμημένα στους κόμβους του cluster , η οποία μπορεί να τεθεί σε επεξεργασία παράλληλα. Τα Rdds ισοδυναμούν με batches . Άρα το Dstream είναι μια ακολουθία από batches όπως φαίνεται και στην παρακάτω εικόνα.

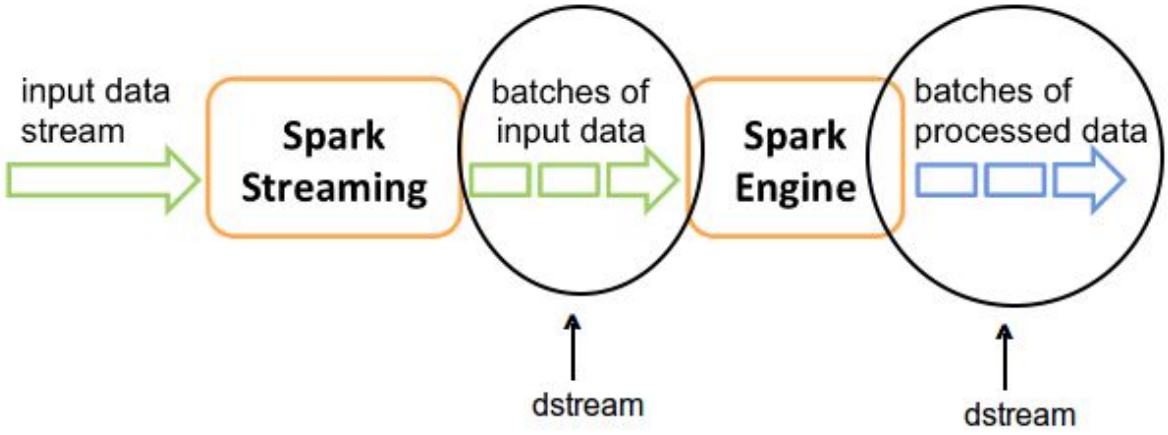


Fig. 10: Dstream: sequence of batches

3.6 Fault tolerance στο Spark Streaming

Στο Spark Streaming τα data τις περισσότερες φορές λαμβάνονται από το δίκτυο (εκτός από όταν χρησιμοποιείται filestream). Για να εξασφαλίσουμε **fault-tolerance** σε αυτή την περίπτωση για όλα τα generated Rdds, τα data που λαμβάνονται αντιγράφονται μεταξύ πολλαπλών Spark executors σε worker nodes του cluster (ο default replication συντελεστής είναι 2). Αυτό οδηγεί σε δύο είδη από data στο system τα οποία πρέπει να ανακτηθούν σε περίπτωση κάποιου σφάλματος.

1) Δεδομένα που έχουν ληφθεί και έχουν αντιγραφεί: Αυτά τα δεδομένα επιβιώνουν από ένα σφάλμα σε έναν worker node καθώς υπάρχει ένα αντίγραφο αυτών σε έναν από τους υπόλοιπους nodes.
2) Δεδομένα που έχουν ληφθεί και βρίσκονται σε buffer για αντιγραφή: Από την στιγμή που δεν έχουν αντιγραφεί , ο μόνος τρόπος για να αποκατασταθούν αυτά τα δεδομένα είναι να τα προσληφθούν ξανά από το source.

Το **checkpointing** στο Spark streaming είναι προαιρετικό. Παρέχει fault-tolerance στα streaming data όταν χρειάζεται . Το Rdd checkpointing είναι απαραίτητο όταν έχουμε stateful transformations.

ΚΕΦΑΛΑΙΟ 4:Flink

4.1 Εισαγωγή

Το Flink είναι ένα processing engine που εκτελεί stateful υπολογισμούς σε οριοθετημένα και μη streams δεδομένων . Το Flink εκτελεί τα batch προγράμματα ως μια ειδική περίπτωση των streaming προγραμμάτων, όπου τα streams είναι οριοθετημένα.

Τα βασικά **building blocks** των Flink programs είναι τα streams και τα transformations. Εννοιολογικά ένα stream είναι δυνητικά μια ατέρμονη ροή από records δεδομένων και το transformation είναι ένα operation που παίρνει ως input ένα η περισσότερα streams και παράγει ένα η περισσότερα streams ως αποτέλεσμα.
Όταν εκτελούνται τα Flink programs αντιστοιχίζονται σε streaming ροές δεδομένων (data flow), οι οποίες αποτελούνται από streams και transformation operators. Κάθε ροή δεδομένων (data flow) ξεκινά με ένα η περισσότερα sources και καταλήγει σε ένα η περισσότερα sinks. Οι ροές δεδομένων μοιάζουν με ακυκλικούς γράφους (Dags). Ένα παράδειγμα streaming ροής δεδομένων φαίνεται στην παρακάτω εικόνα.

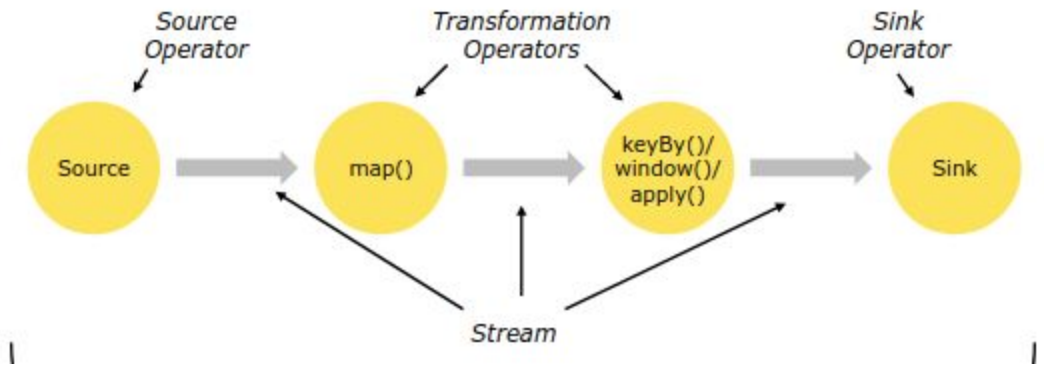


Fig. 10: Streaming Dataflow

Συνήθως υπάρχει 1 προς 1 αντιστοιχία μεταξύ των transformations στα προγράμματα και των operators στις ροές δεδομένων. Κάποιες φορές όμως ένα transformation μπορεί να αποτελείται από πολλαπλούς transformations operators. Στο τρίτο transformation του παραπάνω dataflow έχουμε τρεις operators (keyBy, window, apply).

4.2 Parallel Dataflows

Τα προγράμματα στο Flink είναι εγγενώς παράλληλα και κατανεμημένα. Κατά τη διάρκεια του execution , ένα stream έχει ένα η περισσότερα stream partitions, και κάθε operator έχει ένα η περισσότερα subtasks. Τα operator subtasks είναι ανεξάρτητα το ένα από το άλλο , και εκτελούνται σε διαφορετικά threads και πιθανώς σε διαφορετικά machines η containers.

Ο αριθμός των operator subtasks αποτελεί τον παραλληλισμό του εκάστοτε operator. Ο παραλληλισμός ενός stream είναι πάντα αυτός που υποδεικνύεται από τον operator ο οποίος το δημιούργησε. Διαφορετικοί operators στο ίδιο πρόγραμμα μπορεί να έχουν διαφορετικά επίπεδα παραλληλισμού

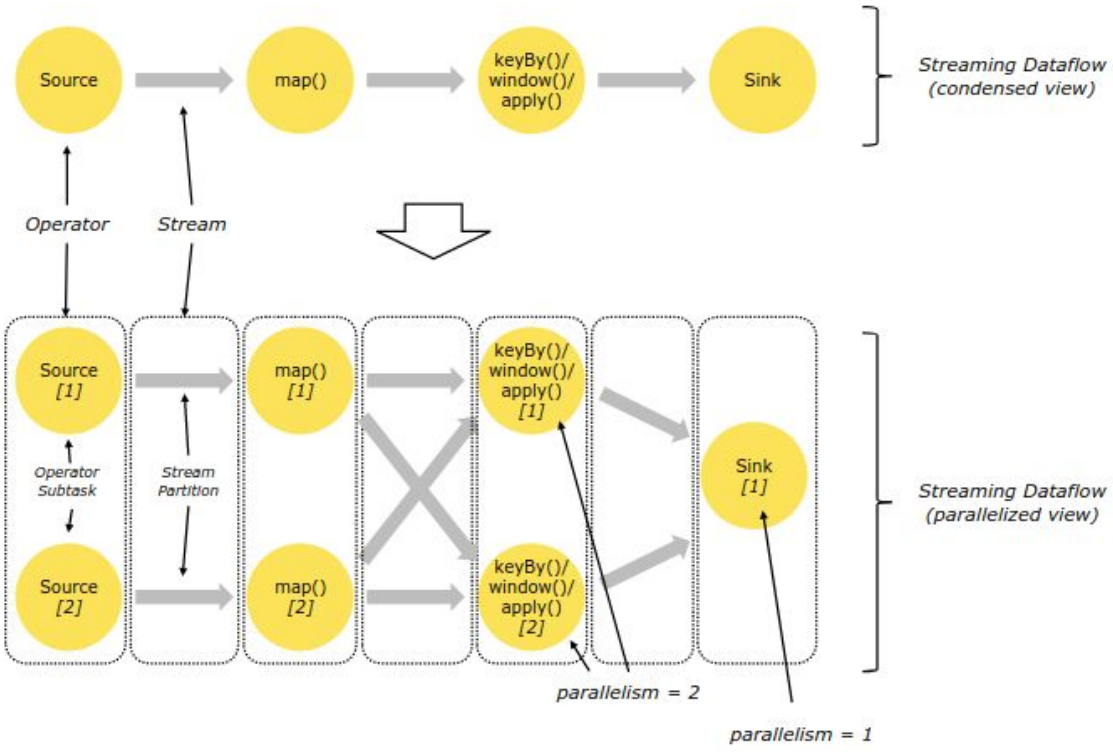


Fig. 11: Parallel Dataflow

4.3 Time

Όσον αφορά τον χρόνο σε ένα streaming πρόγραμμα , μπορούμε να αναφερθούμε σε διαφορετικές έννοιες του χρόνου:

Event Time : Είναι ο χρόνος στον οποίο έχει δημιουργηθεί ένα event. Συνήθως περιγράφεται από ένα timestamp στα events , η οποία έχει ανατεθεί στα events από το producing sensor η το producing service. Στο event time ,η πρόοδος του χρόνου εξαρτάται από τα δεδομένα και όχι από το system clock. Το Flink έχει πρόσβαση σε event timestamps μέσω των timestamps assigners.
Ingestion Time: Είναι ο χρόνος στον οποίο ένα event μπαίνει στη ροή δεδομένων του Flink στο source operator.
Processing Time :Είναι ο τοπικός χρόνος(system time of the machine) σε κάθε operator που εκτελεί ένα time-based operation(like time windows).

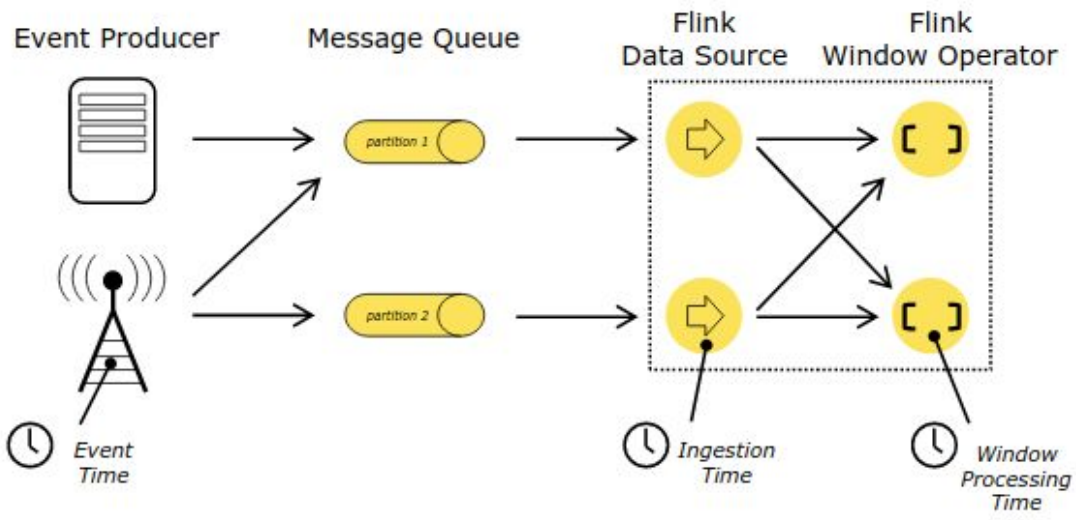


Fig. 12: Notions of time in Flink

4.4 Checkpoints for fault tolerance

Το Flink εφαρμόζει fault tolerance χρησιμοποιώντας ένα συνδυασμό stream replay και σημείων ελέγχου (checkpoints). Κάθε checkpoint σχετίζεται με ένα συγκεκριμένο σημείο του κάθε input stream μαζί με το αντίστοιχο state για κάθε έναν από τους operators. Μια streaming ροή δεδομένων (data flow) μπορεί να ανακτηθεί από ένα checkpoint επαναφέροντας το state των operators και επαναλαμβάνοντας τα events από το σημείο του checkpoint. Το checkpoint interval είναι ένα μέσο συμβιβασμού του overhead του fault-tolerance κατά τη διάρκεια του execution με το recovery time(ο αριθμός των events που πρέπει να επαναληφθούν).
Το checkpointing καθορίζεται από το χρήστη .

4.5 Architecture

Στην παρακάτω εικόνα φαίνεται η αρχιτεκτονική του Flink .

Όταν το Flink ξεκινάει, τίθενται σε λειτουργία ο Job manager και ένας ή και περισσότεροι Task managers. Ο Jobmanager είναι συντονιστής του Flink ενώ οι Task Managers είναι οι workers οι οποίοι εκτελούν parts των παράλληλων προγραμμάτων. Όταν το σύστημα είναι σε local mode ο Job manager και ο Task manager λειτουργούν μέσα στο ίδιο JVM. Όταν το σύστημα είναι σε cluster mode ο Job Manager και ο Task Manager μπορούν να λειτουργούν και σε διαφορετικά JVMs.
Όταν ένα πρόγραμμα υποβάλλεται , δημιουργείται ένας client ο οποίος εκτελεί το preprocessing και μετατρέπει το πρόγραμμα στη μορφή παράλληλου Job Dataflow graph. Ο Job Manager λαμβάνει τον Job Dataflow graph από τον client και είναι υπεύθυνος για τη δημιουργία του execution graph. Στη συνέχεια αναθέτει το job στους Task Managers του cluster και επιβλέπει την εκτέλεση του job. Οι Task Managers τρέχουν τα tasks του job σε ξεχωριστά slots με καθορισμένη παραλληλία. Οι Task Managers στέλνουν το status των tasks που εκτελούν πίσω στον Job Manager.

Η παρακάτω εικόνα απεικονίζει τους παρακάτω ρόλους του συστήματος και τα μεταξύ τους interactions

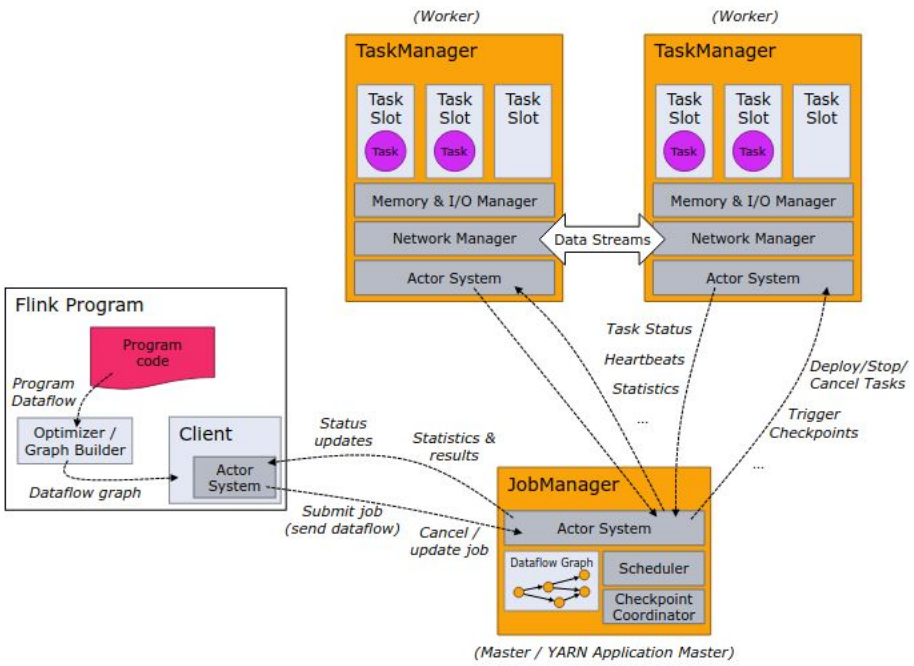


Fig. 13: Flink Architecture

4.6 Scheduling

Τα execution resources του Flink καθορίζονται από τα Task Slots . Κάθε Task Manager έχει ένα η περισσότερα task slots , καθένα από τα οποία μπορεί να τρέξει ένα pipeline από parallel tasks. ΄Ενα pipeline αποτελείται από ένα η περισσότερα διαδοχικά tasks. Το Flink εκτελεί συχνά διαδοχικά tasks παράλληλα. Στη περίπτωση του Streaming αυτό συμβαίνει κάθε φορά.
Η παρακάτω εικόνα απεικονίζει αυτή τη λογική. Ας θεωρήσουμε ένα πρόγραμμα με ένα source δεδομένων , ένα Map Function και ένα Reduce Function. Το source και το MapFunction εκτελούνται με παραλληλία 4 , ενώ το Reduce Function εκτελείται με παραλληλία 3. Ένα pipeline αποτελείται από από την ακολουθία Source-Map -Reduce. Σε έναν cluster με 2 Task Managers με 3 slots ο καθένας , το πρόγραμμα θα εκτελεστεί όπως φαίνεται στην παρακάτω εικόνα.

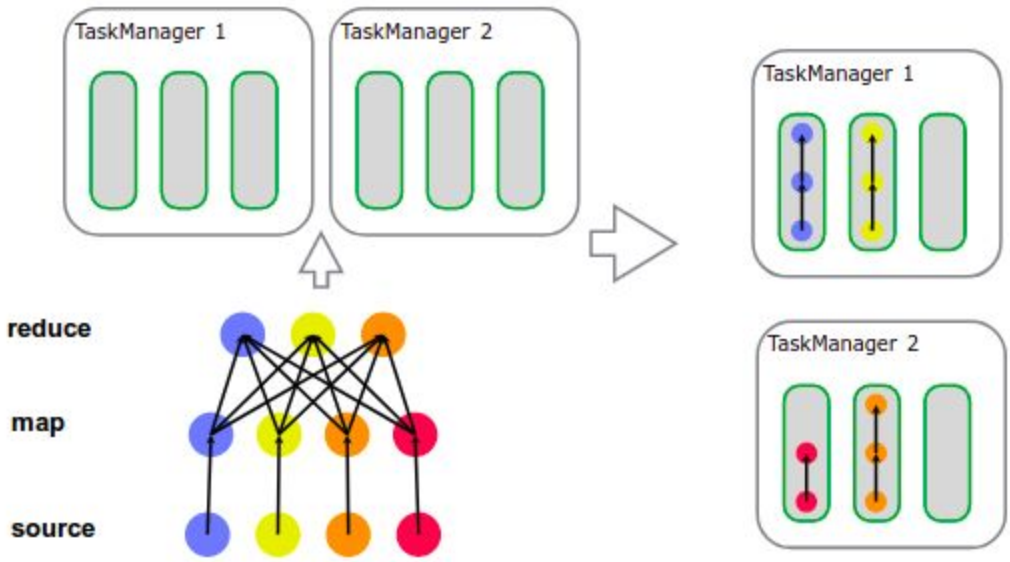


Fig. 14: Scheduling

ΚΕΦΑΛΑΙΟ 5:Kafka

5.1 Εισαγωγή

Ο Kafka είναι ένα ανθεκτικό publish-subscribe messaging system. Ένα messaging system στέλνει μηνύματα ανάμεσα σε processes , applications και servers.

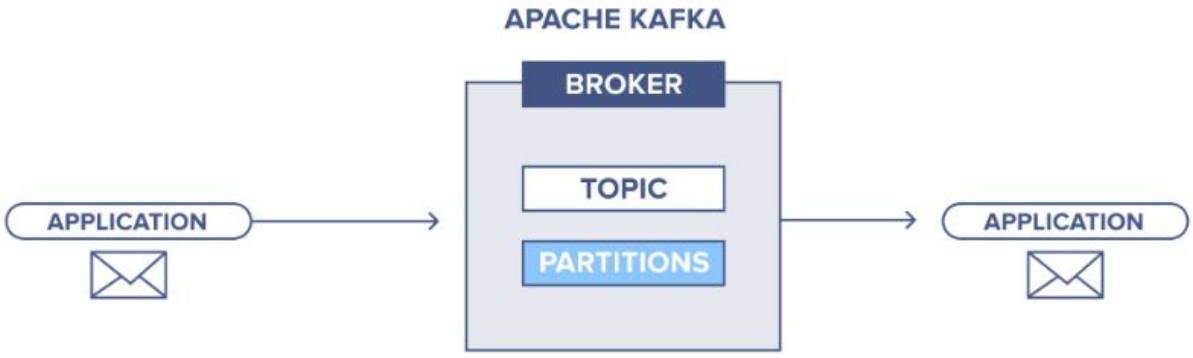


Fig. 15: Kafka: Publish-subscribe messaging system

Το topic αποτελεί το θεμελιώδες abstraction που προσφέρει ο Kafka για ένα stream από records. Το topic είναι μια κατηγορία στην οποία δημοσιεύονται τα records (στον kafka cluster). Applications που συνδέονται με τον Kafka μπορούν να μεταφέρουν ένα record μέσω ενός topic. Ένα record μπορεί να έχει οποιοδήποτε είδους πληροφορίες. Ο Kafka λειτουργεί ως cluster ο οποίος αποθηκεύει stream records στα topics.
Άλλα applications μπορούν να χρησιμοποιήσουν τον Kafka για να επεξεργαστούν η να επανεπεξεργαστούν records από ένα topic. Τα δεδομένα που αποστέλλονται αποθηκεύονται στον kafka cluster μέχρι να λήξει μια περίοδος retention.

Τα records είναι byte arrays που μπορούν να αποθηκεύσουν ένα object σε οποιοδήποτε format. Κάθε record απαρτίζεται από ένα κλειδί , μια τιμή (value) και ένα timestamp.

Τα κύρια μέρη ενός Kafka συστήματος είναι:

Broker: Χειρίζεται όλα τα requests από clients (produce , consume και metadata) και κρατάει τα δεδομένα replicated μέσα στον cluster. Μπορούν να υπάρχουν παραπάνω από ένας broker σε έναν cluster.
Zookeeper: Διατηρεί το state του cluster. (brokers, topics , users)
Producer:Στέλνει records σε έναν broker
Consumer: καταναλώνει batches από records που λαμβάνει από έναν broker

5.2 Kafka Broker

Ένας Kafka Cluster αποτελείται από έναν ή περισσότερους servers (Kafka brokers) (οι οποίοι τρέχουν τον Kafka). Οι Producers είναι processes που προωθούν τα records σε Kafka topics εντός ενός broker. Ένας consumer καταναλώνει/ λαμβάνει τα records από ένα Kafka topic. Είναι εφικτό να έχουμε έναν Kafka broker ωστόσο με αυτόν τον τρόπο δεν εκμεταλλευόμαστε τα πλεονεκτήματα που διαθέτει ο Kafka όπως για παράδειγμα το data replication Η διαχείριση των brokers σε έναν cluster γίνεται από τον Zookeeper. Μπορούν να υπάρχουν πολλαπλοί Zookeepers σε έναν cluster .

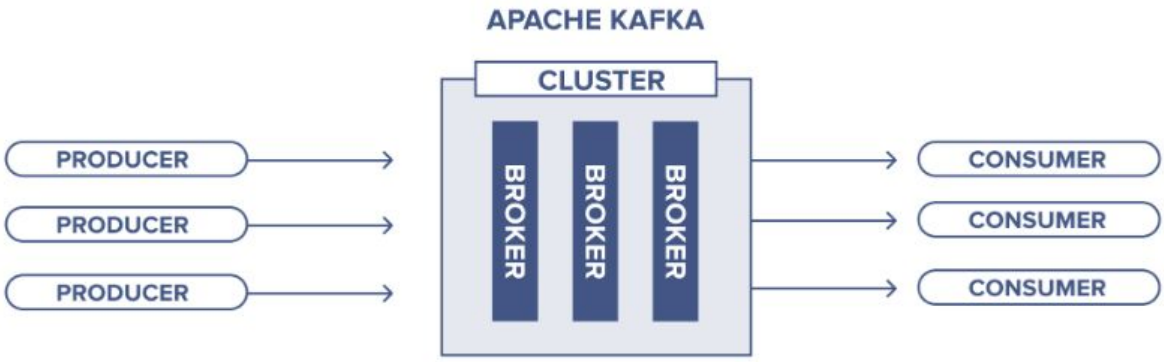


Fig. 16: Kafka overview

5.3 Kafka Topic

Το topic είναι μια κατηγορία στην οποία τα records αποθηκεύονται και γίνονται published(δημοσιεύονται). Όπως προαναφέρθηκε όλα τα Kafka records οργανώνονται σε topics. Producer applications γράφουν δεδομένα σε topics και consumer applications διαβάζουν από τα topics. Ο Kafka cluster διατηρεί όλα τα published records(γίνονται published στον cluster) είτε έχουν καταναλωθεί είτε όχι για ένα συγκεκριμένο χρονικό διάστημα(retention period) το οποίο μπορεί να καθοριστεί από το χρήστη. Για παράδειγμα εάν το retention policy

είναι καθορισμένο στις δύο ημέρες ,τότε για τις επόμενες δύο ημέρες από τη στιγμή που έχει δημοσιευθεί κάποιο record στον kafka cluster(broker), είναι διαθέσιμο για κατανάλωση από κάποιον consumer μέχρις ότου απορριφθεί για απελευθέρωση χώρου.
Ο Kafka διατηρεί τα records σε ένα log , θέτοντας υπεύθυνους τους consumers για την ανίχνευση της θέσης ενός record μέσα στο log. Η θέση ενός record ονομάζεται και offset. Τυπικά ένας consumer διαβάζει τα records γραμμικά (εννοώντας ότι διαβάζει records με αυξανόμενα offsets) αλλά από τη στιγμή που το offset ελέγχεται από τον consumer , ο consumer μπορεί να καταναλώσει τα records με όποια σειρά θέλει. Για παράδειγμα ένας consumer μπορεί να επανέλθει σε ένα παλαιότερο offset για να επανεπεξεργαστεί παλιά δεδομένα ή μπορεί να μεταφερθεί στο πιο πρόσφατο record και να ξεκινήσει να καταναλώνει records από αυτό το σημείο.

5.4 Kafka topic partition

Τα Kafka topics χωρίζονται σε έναν αριθμό από partitions .Το κάθε partition είναι μια ταξινομημένη , αδιάστατη ακολουθία από records.Για κάθε topic , ο Kafka cluster διατηρεί ένα κατανεμημένο log το οποίο φαίνεται στην παρακάτω εικόνα. Σε κάθε record που ανήκει σε κάποιο partition ανατίθεται ένας sequentail αριθμός id που ονομάζεται offset ο οποίος διαχωρίζει κάθε record από τα υπόλοιπα μέσα στο partition. Τα partitions επιτρέπουν στα topics να παραλληλοποιούνται με το να διαχωρίζουν τα δεδομένα ενός topic σε πολλαπλούς brokers. Τα topics στον kafka είναι multi-subscriber. Αυτό σημαίνει ότι ένα topic μπορεί να έχει μηδέν , ένα η και πολλούς consumers που καταναλώνουν τα data που είναι γραμμένα σε αυτό.

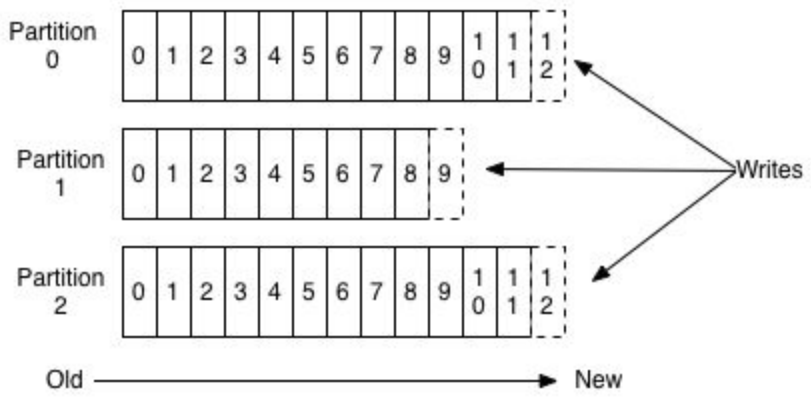


Fig. 16: Anatomy of a Topic

Στον Kafka το replication εφαρμόζεται στο επίπεδο του partition. Κάθε partition έχει ένα η περισσότερα αντίγραφα εννοώντας ότι τα partitions περιέχουν messages τα οποία έχουν αντίγραφα σε άλλους Kafka brokers του cluster. Κάθε partition(αντίγραφο) έχει έναν server που

λειτουργεί ως leader και οι υπόλοιποι λειτουργούν ως followers. Το leader αντίγραφο διαχειρίζεται όλα τα read-write requests για ένα συγκεκριμένο partition και οι followers αντιγράφουν τον leader. Αν ο επικεφαλής leader τεθεί εκτός λειτουργίας λόγω σφάλματος τότε ένας από τους follower server γίνεται ο leader. Όταν ένας producer κάνει publish ένα record σε ένα topic , το δημοσιοποιεί στον leader. Ο leader προσαρτά το record στο αντίστοιχο log και αυξάνει το record offset του log. Ο kafka καθιστά διαθέσιμο ένα record προς κατανάλωση από κάποιον consumer όταν το record προσαρτηθεί σε κάποιο partition και του ανατεθεί ένα offset.

Οι Producers δημοσιεύουν (publish) τα δεδομένα στα topics της επιλογής τους. Ο Producer επιλέγει σε ποιο partition θα ανατεθεί ένα stream record (δεν εξαρτάται από τον broker). Αυτό μπορεί να γίνει είτε με round-robin τρόπο είτε σύμφωνα με κάποια partition συνάρτηση. Ο producer βάζει ένα κλειδί σε κάθε record καθορίζοντας με αυτό τον τρόπο σε ποια partition πρέπει να πάει το κάθε record. Όλα τα records με το ίδιο κλειδί πηγαίνουν στο ίδιο partition. Πριν ο producer στείλει κάποιο record σε κάποιο partition , πρέπει να ζητήσει metadata για τον cluster από τον broker. Τα metadata περιέχουν πληροφορίες σε ποιον broker είναι ο leader της κάθε partition . Αυτές τις πληροφορίες τις χρειάζεται ο producer καθώς γράφει τα records πάντα στον partition leader. Ο producer στη συνέχεια χρησιμοποιεί το key του record για να ξέρει σε ποιο partition να γράψει. Το default implementation είναι να χρησιμοποιηθεί το hash του κλειδιού για να υπολογιστεί το partition στο οποίο θα γραφεί το record

5.5 Consumers

Ο Consumer (περισσότερο γνωστό και ως consumer groups) αποτελείται από έναν ή περισσότερους consumers. Το consumer group δημιουργείται προσθέτοντας το property '**group id**' σε έναν consumer. Δίνοντας το ίδιο group id σε έναν άλλο consumer σημαίνει ότι θα μοιράζονται το ίδιο group με τον αρχικό consumer. Ο broker θα κατανείμει τα records ανάλογα με το ποιος consumer πρέπει να διαβάσει από ποια partitions και επίσης καταγράφει για το group σε ποιο offset βρίσκεται η κάθε partition. Έχει την δυνατότητα να κάνει αυτή την καταγραφή καθώς όλοι οι consumers ενημερώνουν (τον broker) ποιο offset έχουν χειριστεί. Τα records κατανέμονται στους consumers που ανήκουν στο ίδιο consumer group. Κάθε φορά που ένας consumer προστίθεται η αποβάλλεται από ένα group , η κατανάλωση των records εξισορροπείται από τους υπόλοιπους consumers που ανήκουν στο ίδιο group.
Οι consumers διαβάζουν τα records από topic partitions ξεκινώντας από ένα συγκεκριμένο offset και τους επιτρέπεται να διαβάσουν από οποιοδήποτε offset point επιλέξουν. Αυτό επιτρέπει στους consumers να κάνουν join στον cluster οποιαδήποτε στιγμή .
Κάνοντας χρήση των consumer groups , οι consumers μπορούν να παραλληλοποιηθούν έτσι ώστε πολλαπλοί consumers να μπορούν να διαβάζουν από πολλαπλά partitions ενός topic επιτρέποντας έτσι ένα μεγάλο message throughput. Ο αριθμός των partitions επηρεάζει τον μέγιστο αριθμό παραλληλίας των consumers καθώς δεν μπορούν να υπάρχουν περισσότεροι consumers από partitions

5.6 Tuning Kafka performance

Υπάρχουν δύο παράμετροι του kafka οι οποίοι είναι ιδιαίτερα σημαντικοί για τη βελτίωση του Kafka performance. Αυτές οι παράμετροι είναι το batch size και το linger.ms και καθορίζονται στον producer.

Ο producer αποτελείται από ένα pool από buffer space το οποίο κρατάει records τα οποία ακόμα δεν έχουν μεταφερθεί στον kafka server όπως επίσης και ένα background I/O thread το οποίο είναι υπεύθυνο για να μετατρέπει αυτά τα records σε requests και να τα μεταφέρει στον cluster .Το send() method είναι ασύγχρονο . Όταν καλείται προσθέτει records σε έναν buffer που έχει pending records(δηλαδή τα records στον buffer περιμένουν να παραδοθούν στον cluster) . Αυτό επιτρέπει στον producer να κάνει batch σε μεμονωμένα records για αποδοτικότητα. Ο producer διατηρεί buffers από records που δεν έχουν σταλεί για κάθε partition. Αυτοί οι buffers έχουν μέγεθος το οποίο καθορίζεται από το batch size **config**. Μεγαλώνοντας το batch size έχει ως αποτέλεσμα περισσότερο batching αλλά χρειαζόμαστε και περισσότερο memory.

By default ένας buffer είναι διαθέσιμος να στείλει αμέσως ακόμα και αν υπάρχει αχρησιμοποίητο space στον buffer. Παρόλα αυτά εάν θέλουμε να μειώσουμε τα requests μπορούμε να θέσουμε το **linger.ms** μεγαλύτερο του μηδενός. Αυτό θα δώσει την εντολή στον producer να περιμένει τόσα ms όσα και η τιμή του linger.ms μέχρι να στείλει ένα request ελπίζοντας ότι περισσότερα records θα φτάσουν για να γεμίσουν το ίδιο batch. Αυτό το setting δίνει το άνω όριο στη καθυστέρηση του batching. Όταν όμως φτάσουμε το batch size για ένα partition , το batch θα σταλεί ανεξάρτητα από την τιμή του linger. Εάν όμως έχουν συγκεντρωθεί λιγότερα bytes σε σχέση με το batch size για ένα partition , θα χρονοτριβούμε μέχρι να τελειώσει το διάστημα του linger περιμένοντας να εμφανιστούν άλλα records.Η default τιμή του linger.ms είναι 0 (δεν υπάρχει καθυστέρηση) .Για παράδειγμα θέτοντας το linger.ms=5 ,μειώνουμε τον αριθμό των requests που στέλνονται στον cluster, προσθέτοντας 5ms latency περιμένοντας περισσότερα records εάν δεν έχει γεμίσει ο buffer. Άρα για μεγαλύτερο **latency** και μεγαλύτερο **throughput** αυξάνουμε το linger.ms. Στη γενική περίπτωση records τα οποία φτάνουν στον buffer χρονικά κοντά, θα ομαδοποιηθούν και θα σταλούν μαζί ακόμα και εάν ισχύει linger.ms=0.

Ανακεφαλαιώνοντας ένα batch έχει ολοκληρωθεί είτε εάν φτάσει σε ένα συγκεκριμένο size (batch size) είτε εάν λήξει ένα συγκεκριμένο χρονικό διάστημα (linger.ms). Το batch size και το linger.ms καθορίζονται στον producer. Το default για το batch size είναι 16,384 bytes και το default για το linger.ms είναι 0 ms. Όταν το batch φτάσει το batch size η εκπνεύσει ο χρόνος του linger.ms , το σύστημα θα στείλει το batch στον kafka cluster.

Η αύξηση του batch size και του linger.ms έχει ως αποτέλεσμα την αύξηση του throughput αλλά και του latency. Ανάλογα με την εφαρμογή μπορούμε να καθορίσουμε τις τιμές αυτών των δύο παραμέτρων έτσι ώστε η αύξηση του throughput να επισκιάζει την αύξηση του latency.

5.7 Compression στον Kafka

Το compression βοηθάει στην μείωση της ποσότητας των data που αποθηκεύονται στο δίσκο η στέλνονται στο δίκτυο. Το compression αποτελεί ένα trade-off μεταξύ περισσότερου IO η περισσότερης χρησιμοποίησης της CPU. Στην περίπτωση των δικών μας πειραμάτων προτιμήσαμε το compression καθώς τα applications που τρέχουμε είναι IO intensive και έτσι το CPU usage είναι αρκετά μικρότερο σε σχέση με την πυκνότητα των IO operations. Τα applications που τρέχουμε χρησιμοποιούν κατά κόρον το δίκτυο για μεταφορά δεδομένων και έτσι το compression κρίνεται απαραίτητο για την βελτίωση του performance.

Όταν μειώνεται το μέγεθος των messages που στέλνει ο kafka producer στον broker, χρησιμοποιείται λιγότερο bandwidth και ως εκ τούτου μπορούμε να αυξήσουμε τον συνολικό αριθμό των messages που στέλνονται στους brokers. Πέρα από την αύξηση του συνολικού αριθμό των records που μπορούμε να στείλουμε στον broker αυξάνεται και το throughput με το οποίο στέλνουμε τα messages στο broker. Αυτό σημαίνει ότι μειώνεται ο χρόνος που περιμένει το engine (Spark ή Flink) για να του έρθουν στοιχεία από τον kafka για να καταναλώσει και έτσι συνολικά βελτιώνεται το performance.Η συμπίεση των messages γίνεται στον producer και η αποσυμπίεση στον kafka consumer.

5.7.1 Record batch

Τα messages ή records γράφονται πάντα στα batches. Ο τεχνικός όρος για ένα batch από messages είναι το record batch , η δομή του οποίου φαίνεται στην παρακάτω εικόνα. Το record batch περιέχει ένα η περισσότερα records και ένα record batch overhead section. Το record batch overhead περιέχει metadata για τα records

Όταν χρησιμοποιούμε συμπίεση μειώνουμε το size του κάθε record batch. Ο producer συμπιέζει μόνο τα records μέσα στο batch και δεν συμπιέζει το record batch overhead . Όταν είναι ενεργοποιημένο το compression , τοποθετείται ένα flag bit για τον τύπο του compression στο record batch overhead. Αυτό το flag θα χρησιμοποιηθεί κατά τη διάρκεια του decompression από τον consumer.

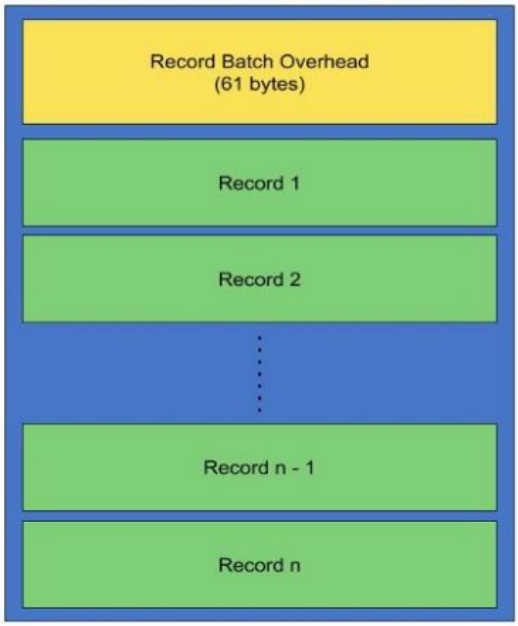


Fig. 17: Record Batch

Ο Kafka υποστηρίζει διάφορους αλγόριθμους συμπίεσης. Το gzip compression έχει μεγαλύτερο compression ratio οδηγώντας σε μεγαλύτερη χρησιμοποίηση της CPU. Το Snappy έχει μικρότερο compression ratio και μικρότερη χρησιμοποίηση της CPU. Στα πειράματα μας χρησιμοποιούμε snappy compression. Το compression algorithm καθορίζεται από το compression.type config στον producer.

ΚΕΦΑΛΑΙΟ 6:Εκτέλεση

6.1 okeanos

Για την διεξαγωγή των πειραμάτων δημιουργήσαμε έναν cluster χρησιμοποιώντας resources από τον **okeano knosso.** Ο cluster αποτελείται από 7 machines. Τα πρώτα τρία machines έχουν οκταπύρηνη CPU και 16GB RAM , τα τρία επόμενα machines έχουν τετραπύρηνη CPU και 8GB RAM και το τελευταίο machine έχει οκταπύρηνη CPU και 8 GB RAM. Ένα από τα τρία machines με 8πύρηνη CPU και 16GB RAM έχει ανατεθεί στον driver του Spark και στον Job Manager του Flink αντίστοιχα. Ένα ακόμη machine με 8πύρηνο CPU και 16GB RAM έχει ανατεθεί στον kafka server και τα υπόλοιπα machines αποτελούν τα worker nodes του Spark και τα Task Managers του Flink αντίστοιχα. Η **αντιστοιχία** των execution cores του Spark και του Flink είναι 1 προς 1 με τα cores των machines του cluster. Συγκεκριμένα στο Spark ένας executor core ένος worker (node) αντιστοιχεί σε έναν core ενος machine και στο Flink ένα task slot αντιστοιχεί σε ένα core ενός machine.

Το **default setup** για τη διεξαγωγή των πειραμάτων είναι 4GB RAM στον driver του Spark και στον Job manager του Flink αντίστοιχα και 4GB RAM σε κάθε worker του Spark και task manager του Flink αντίστοιχα.
Στα πειράματα του join operator χρησιμοποιήσαμε 8GB RAM στον driver του Spark και στον Job manager του Flink αντίστοιχα και 6GB RAM σε κάθε worker του Spark και task manager του Flink αντίστοιχα.

6.2 Τοπολογία

Στο παρακάτω σχήμα φαίνεται ο τρόπος με τον οποίο λειτουργούν τα stream και micro-batch processing engines και κατά συνέπεια η αλληλουχία των ενεργειών που διεξάγεται στα πειράματα μας . Στην περίπτωση μας διοχετεύουμε πολύ μεγάλα αρχεία στο message queue (kafka server) του cluster . Με αυτό τον τρόπο τα στοιχεία των αρχείων αντιμετωπίζονται ως stream records. Στη συνέχεια τα records που βρίσκονται μέσα στο message queue εισάγονται στο εκάστοτε processing engine(Spark ή Flink) και υφίστανται transformations ανάλογα με τον κώδικα της εκάστοτε streaming application . Τέλος το output των transformations μπορεί να είναι ένα καινούριο stream , είτε να αποθηκευτεί σε κάποια βάση δεδομένων ή σε κάποιο filesystem. Στην περίπτωση μας το τελικό **output** των transformations που εκτελούνται πάνω στα stream records που εισάγουμε στον kafka , αποθηκεύεται κατανεμημένα στα file systems των machines του cluster.

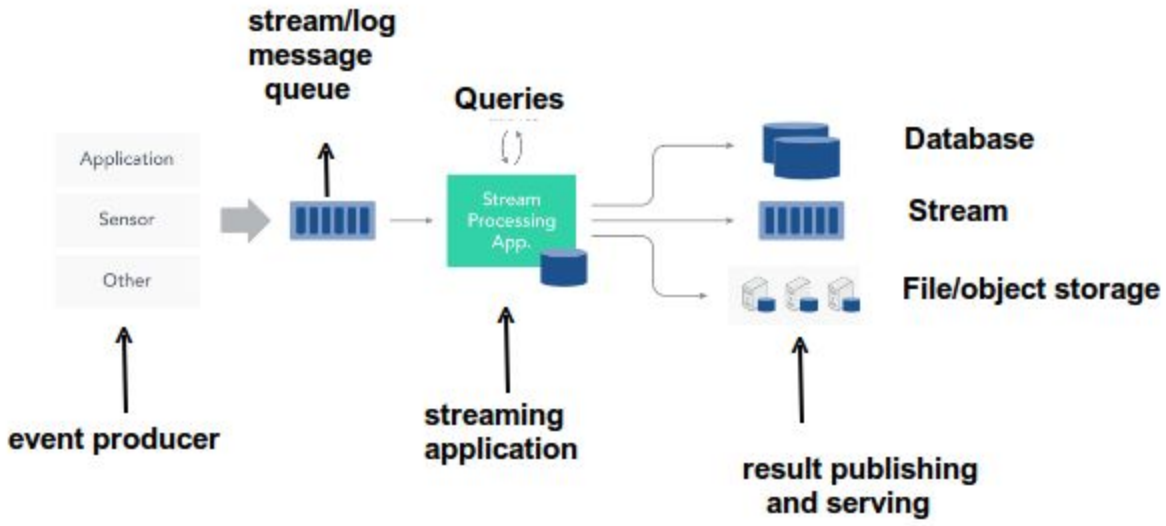


Fig. 18: Stream Processing for Batch/Real-Time Data Processing, and Event-driven Applications

6.3 Operators

Στα πειράματα συγκρίνουμε το performance μεταξύ του Spark streaming και του Flink σε σχέση με κάποιους operators. Οι operators που χρησιμοποιούμε για αυτή τη σύγκριση είναι το filter, το window join και το window aggregate by key.Τα implementations των operators τα βρήκαμε online αλλάζοντας κάποιες παραμέτρους ανάλογα με το application ωστόσο διαμορφώσαμε τον τρόπο με τον οποίο το Spark streaming και το Flink λειτουργούν ως kafka consumers . Τα applications στο Flink είναι γραμμένα σε java ενώ τα applications στο Spark είναι γραμμένα σε scala.

filter	Φιλτράρει records ενός stream που πληρούν μια συγκεκριμένη συνθήκη που ορίζεται από το χρήστη. Τα υπόλοιπα records που δεν πληρούν την συνθήκη απορρίπτονται.
window aggregate by key	Εφαρμόζεται σε records ενός stream που έχουν τη μορφή **(key,value)**.Προσθέτει τις τιμές των records που έχουν κοινό κλειδί και ανήκουν στο ίδιο window.
window join	Όταν έχουμε δύο input streams όπου το πρώτο stream έχει records της μορφής **(key,V)** και το δεύτερο stream records της μορφής **(key, W)** επιστρέφεται ένα νέο stream με records της μορφής **(key, (V,W))** που ανήκουν στο ίδιο window **.** Για κάθε key περιλαμβάνονται όλοι οι δυνατοί συνδυασμοί **(V,W)** .

TABLE I: Operators Definition

6.4 Τι είναι το window

Το **window** είναι ένα χρονικά ορισμένο interval που δέχεται stream records μέχρι τη λήξη αυτού του interval. Τα transformations πραγματοποιούνται πάνω σε records που ανήκουν στο ίδιο window.

Σε ένα **tumbling window** τα records ομαδοποιούνται σε ένα window που είναι χρονικά ορισμένο (ή ορισμένο με βάση κάποιον counter). Κάθε record ανήκει μόνο σε ένα window. Τα tumbling windows έχουν ένα καθορισμένο μέγεθος και δεν αλληλοεπικαλύπτονται όπως φαίνεται και στην παρακάτω εικόνα. Για παράδειγμα αν ορίσουμε ένα tumbling window με μέγεθος 5 λεπτά , το τρέχον window θα υπολογιστεί και ένα καινούριο window θα ξεκινάει κάθε πέντε λεπτά όπως φαίνεται και στην παρακάτω εικόνα.

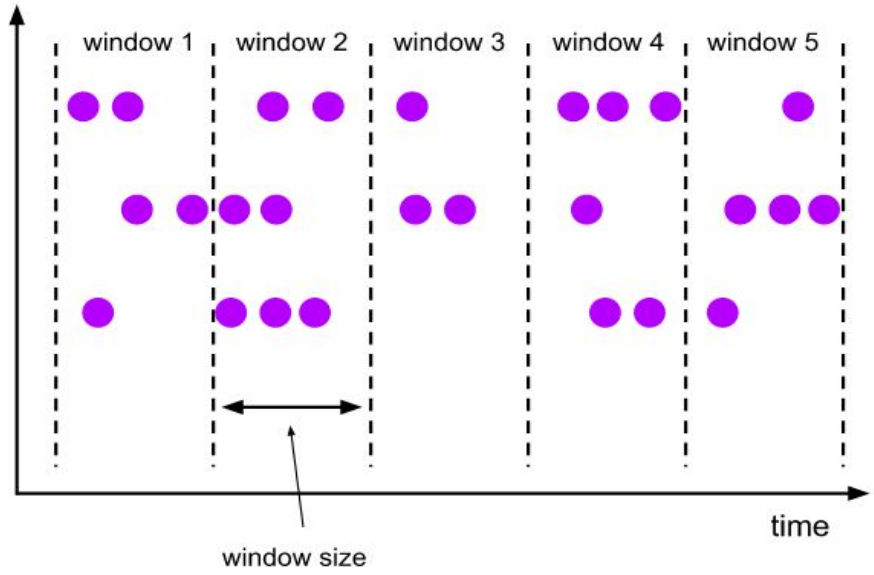


Fig. 19: Tumbling Window

Σε ένα **sliding window** , τα records ομαδοποιούνται σε ένα window το οποίο ολισθαίνει (τσουλάει) κατά μήκος του data stream σύμφωνα με ένα καθορισμένο interval. Όπως και στο tumbling window , το μέγεθος του window είναι καθορισμένο . Μια επιπρόσθετη **window slide** παράμετρος ελέγχει πόσο συχνά ένα sliding window ξεκινάει. Ως εκ τούτου , τα sliding windows μπορούν να είναι αλληλοεπικαλυπτόμενα εάν το slide είναι μικρότερο από το μέγεθος του window . Σε αυτή την περίπτωση τα records ανατίθενται σε πολλαπλά windows όπως φαίνεται και στην παρακάτω εικόνα.

Για παράδειγμα , θα μπορούσαμε να έχουμε windows με μέγεθος 10 λεπτά που ολισθαίνουν για 5 λεπτά. Έτσι κάθε πέντε λεπτά παίρνουμε ένα window που περιέχει records που έφτασαν κατά τη διάρκεια των τελευταίων 10 λεπτών όπως φαίνεται και στην παρακάτω εικόνα.

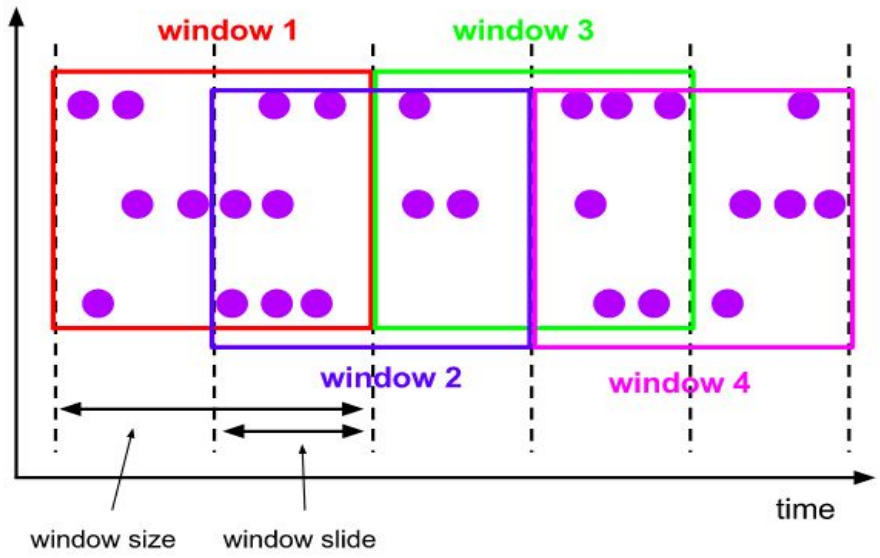


Fig. 20: Sliding Window

6.5 Επιλογή batch size στο Spark Streaming

Για να είναι ένα Spark Streaming application που τρέχει σε ένα cluster σταθερό , το σύστημα θα πρέπει να είναι σε θέση να επεξεργάζεται δεδομένα με την ίδια ταχύτητα με την οποία λαμβάνονται. Τα batches θα πρέπει να μπορούν να τίθενται σε επεξεργασία με την ίδια ταχύτητα με την οποία δημιουργούνται . Με άλλα λόγια το batch processing time θα πρέπει να είναι μικρότερο από το batch interval (το οποίο διαπιστώνεται στο Web UI).
Ανάλογα με την εφαρμογή το batch interval μπορεί να έχει μια σημαντική επίδραση στα data rates τα οποία μπορούν να διατηρηθούν από το application σε ένα fixed σύνολο από cluster resources.Για παράδειγμα για ένα συγκεκριμένο data rate , το σύστημα ενδέχεται να μπορεί να συμβαδίσει με ένα batch interval των 2 second και όχι με ένα batch interval των 500 ms. Έτσι το batch interval πρέπει να τεθεί με τέτοιο τρόπο έτσι ώστε το αναμενόμενο data rate να μπορεί να διατηρηθεί από το application.

Μια καλή προσέγγιση για την επιλογή του σωστού batch interval για ένα application είναι να τεστάρουμε το application με ένα safe batch interval (5-10 seconds) και ένα χαμηλό data rate.Για να βεβαιωθούμε ότι το σύστημα μπορεί να συμβαδίσει με το data rate συγκρίνουμε το processing time του κάθε batch με το batch interval του. Αν το processing time είναι μικρότερο από το batch interval τότε το σύστημα είναι σταθερό.Αλλιώς αν το processing time υπερβαίνει το batch interval και συνεχώς αυξάνεται τότε το σύστημα δεν μπορεί να συμβαδίσει και έτσι είναι unstable. Από τη στιγμή που εξασφαλίζεται ένα stable configuration , ένας χρήστης μπορεί να δοκιμάσει να αυξήσει το data rate η να μειώσει το batch size. Η στιγμιαία αύξηση στο delay

εξαιτίας μιας προσωρινής αύξησης στο data rate μπορεί να μην αποτελεί πρόβλημα εφόσον το delay μειώνεται ξανά και επιστρέφει σε μια τιμή μικρότερη από το batch size

Στην περίπτωση μας στον filter operator επιλέξαμε το **batch interval να είναι 10s**. Διαπιστώνουμε ότι με αυτό το batch interval εξασφαλίζεται ένα stable configuration (processing time<batch interval). Στη συνέχεια αυξάνουμε το data rate αυξάνοντας τον αριθμό των kafka partitions των input streams και βρίσκουμε τον αριθμό των kafka partitions των input streams στον οποίο το filter application έχει το μικρότερο latency και συνεπώς επιτυγχάνεται το sustainable throughput. Γενικά όσα περισσότερα partitions έχει ένα kafka stream , τόσο μεγαλύτερο είναι το throughput με το οποίο στέλνεται το stream σε ένα streaming application.

Στο aggregate by key έχουμε ένα window 20s με sliding interval 10s . Το batch interval σε αυτή την περίπτωση πρέπει να είναι μικρότερο ή ίσο από το sliding interval για να έχουμε σωστά αποτελέσματα. Επιλέγουμε **batch size 10s .** Διαπιστώνουμε ότι με αυτό το batch interval εξασφαλίζεται ένα stable configuration (processing time<batch interval).Στη συνέχεια αυξάνουμε το data rate αυξάνοντας τον αριθμό των kafka partitions των input streams και βρίσκουμε τον αριθμό των kafka partitions των input streams στον οποίο το aggregate by key application έχει το μικρότερο latency και συνεπώς επιτυγχάνεται το sustainable throughput. Το batch size των 10s εξασφαλίζει πιο stable configuration σε σχέση με τα 5s για batch interval.

Στο window join επιλέγουμε το batch size να είναι ίδιο με το window size για λόγους που έχουμε εξηγήσει στο 9.4.

6.6 Data generation

Όλα τα input streams που χρησιμοποιούμε στα Spark και Flink applications στον cluster είναι αρχεία που έχουν παραχθεί από **data generator**. Δεν μπορούμε να χρησιμοποιήσουμε data generator στον cluster για την δημιουργία stream records καθώς το σύστημα πέφτει λόγω έλλειψης σε resources. Για αυτό το λόγο παράγουμε εξωτερικά από τον cluster **αρχεία** που στη συνέχεια διοχετεύονται στον kafka server του cluster και ισοδυναμούν με streams. Συγκεκριμένα παράγουμε αρχεία από data μέσω δύο applications σε ένα **localhost**.

Το πρώτο application είναι ένας **event producer** ο οποίος δημιουργεί events/records τα οποία αποστέλλει μέσω ενός kafka producer σε έναν local kafka server**(message queue)** . Στη συνέχεια το δεύτερο **application** που είναι ουσιαστικά ένας kafka consumer , καταναλώνει τα records που λαμβάνει και τα τυπώνει σε μορφή αρχείου σε ένα local **filesystem**.

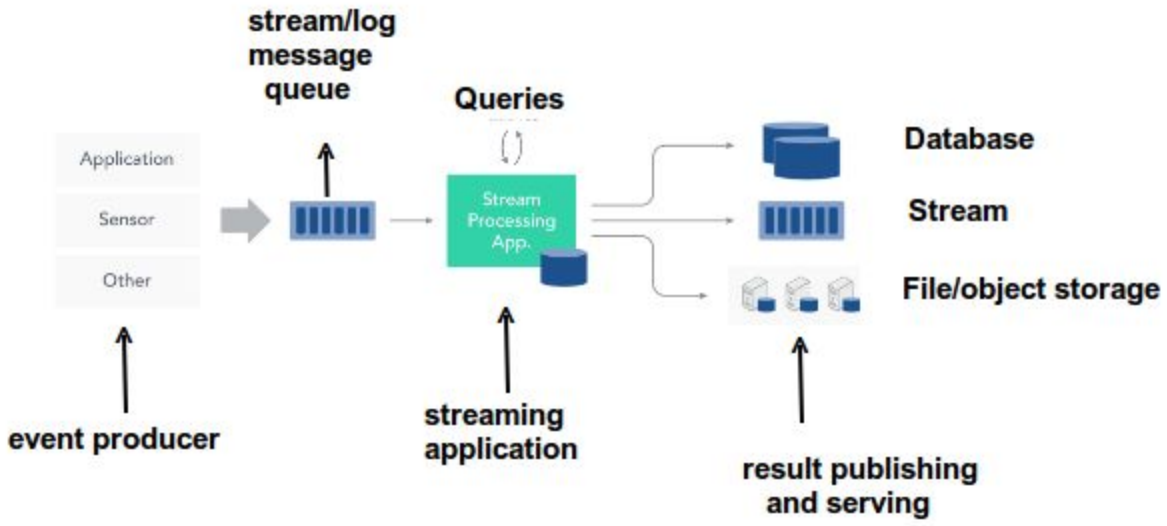


Fig. 21: Stream Processing for Batch/Real-Time Data Processing, and Event-driven Applications

Κάθε **αρχείο** που παράγουμε με τον παραπάνω τρόπο το στέλνουμε στον cluster και χρησιμοποιείται ως inputstream για κάποιο πείραμα που διεξάγουμε στον cluster όπως περιγράφηκε στην ενότητα **6.2** .Και τα δύο applications είναι γραμμένα σε scala.

(Συγκεκριμένα ο kafka producer του kafka server που διαθέτει ο cluster δέχεται ένα αρχείο από αυτά που έχουμε παράξει και στέλνει το κάθε record του αρχείου που διαβάζει στον kafka broker προκειμένου αυτά τα records να καταναλωθούν στη συνέχεια από Spark και Flink applications που λειτουργούν ως kafka consumers)

6.7 Παράμετροι εξέτασης

Στα **streams** που χρησιμοποιούνται ως input για τα Spark και Flink applications έχουμε παίξει με διάφορες παραμέτρους . Συγκεκριμένα για όλους τους operators που χρησιμοποιούμε στα πειράματα εισάγαμε διαδοχικά διαφορετικά input streams . Τα διαφορετικά input streams διαφοροποιούνται μεταξύ τους σε σχέση με ένα πλήθος παραμέτρων . Οι **παράμετροι** αυτοί είναι το **selectivity** , ο αριθμός των κλειδιών και ο αριθμός των διαστάσεων του key όταν αναφερόμαστε σε keyed streams. Το selectivity στο join operator έχει διαφορετική έννοια από το selectivity στο filter operator. Το join δέχεται ως input δύο διαφορετικά streams.

Selectivity στο join	Είναι ο λόγος του αθροίσματος των records του πρώτου stream και των records του δεύτερου stream που έχουν μεταξύ τους κοινά κλειδιά (τα records του πρώτου με τα records του δεύτερου) προς τον συνολικό αριθμό των records των δύο input streams. Ουσιαστικά είναι ένας δείκτης που δείχνει πόσο πυκνό είναι το operation του join. Το selectivity στο join καθορίζεται από τα input streams και την μεταξύ τους σχέση.
Selectivity στο filter	Είναι ο λόγος του αθροίσματος των records του inputstream που πληρούν τη συνθήκη του filter προς το συνολικό αριθμό των records του input stream. Το selectivity σε αυτή την περίπτωση καθορίζεται από το condition του filter.
Αριθμός των κλειδιών	Είναι ο αριθμός των διαφορετικών κλειδιών σε ένα inputstream με records της μορφής **(key,value)**
Αριθμός διαστάσεων του key	Είναι ο αριθμός των διαστάσεων του key σε ένα keyed input stream. Για παράδειγμα ένα stream με records της μορφής **(key,value)** , έχει κλειδί μιας διάστασης . Ένα stream με records της μορφής **((key1,key2), value)** ,έχει κλειδί δύο διαστάσεων.

TABLE II: Definition of parameters

6.8 Χρήση Kafka στη διεξαγωγή πειραμάτων

Χρησιμοποιήσαμε apache kafka για την διεξαγωγή των πειραμάτων .Διαλέξαμε τον **kafka** καθώς είναι το πιο δημοφιλές messaging system. Αυτό οφείλεται στα πολλά πλεονεκτήματα που διαθέτει ο Kafka όπως η αξιοπιστία, η κλιμακωσιμότητα , η ανθεκτικότητα κ.α .

Συγκεκριμένα ο Kafka μπορεί να δουλέψει εύκολα με ένα τεράστιο όγκο από data streams. Είναι fault-tolerant, έχει ανθεκτικότητα καθώς χρησιμοποιεί κατανεμημένα commit logs που σημαίνει ότι τα messages αποθηκεύονται στο δίσκο όσο πιο γρήγορα γίνεται. Ο Kafka μπορεί να χειριστεί το scalability και στις τέσσερις διαστάσεις, εννοώντας στους event producers, event processors, event consumers και event connectors. Με άλλα λόγια ο Kafka κλιμακώνει εύκολα χωρίς διακοπές. Ο kafka έχει υψηλό performance καθώς είτε κάνει publish είτε subscribe τα

messages , διατηρεί υψηλό throughput. Διαθέτει επεκτασιμότητα καθώς υπάρχουν πολλοί τρόποι με τους οποίους τα διάφορα applications μπορούν να συνδεθούν και να χρησιμοποιήσουν τον Kafka.
Μπορούμε να αυξήσουμε το throughput των input streams που δέχονται οι kafka consumers στο Spark και το Flink αντίστοιχα πειράζοντας τις μεταβλητές **linger.ms** και **batch size** στον producer του kafka όπως έχουμε περιγράψει στο κεφάλαιο 5 . Το linger.ms καθορίζει το χρόνο που περιμένει ο producer μέχρι να στείλει το batch στον kafka cluster έτσι ώστε το batch να γίνει διαθέσιμο στους consumers. Το batch size είναι by default σε bytes και πειράζοντας την τιμή του καθώς και την τιμή του linger καθορίζουμε το throughput του producer . Για παράδειγμα αν το linger είναι 5000 ms και το batch size 13.000.000(12,4 Mbytes) , τότε ο producer θα πρέπει να περιμένει 5000 ms μέχρι να στείλει το batch των 12,4 Mbytes . Υπάρχει περίπτωση στα 5000 ms να έχουμε λιγότερα bytes από το batch size. Τότε ο producer θα στείλει τα bytes που έχει μαζέψει και ας είναι λιγότερα από το batch size. Σε περίπτωση που ο buffer γεμίσει με τόσα bytes όσα και το batch size πριν τελειώσει η καθυστέρηση που εισάγει το linger , το batch θα σταλεί κανονικά στον kafka cluster χωρίς να περιμένει να τελειωσει η καθυστέρηση που εισάγει το linger.

6.9 Spark processing time

Όλα τα πειράματα είναι σε **processing time** καθώς το spark streaming δουλεύει by default σε processing time . Το Flink δουλεύει και σε event time και σε ingestion time .
Όλα τα operations του Spark Streaming ανεξαρτήτως του αν έχουν windows ή όχι θεωρούνται time-based operations.Αυτό συμβαίνει διότι το Spark streaming δουλεύει με **batches**. Το batch interval καθορίζει το μέγεθος του batch σε seconds. Δηλαδή κάθε batch οριοθετείται χρονικά. Επειδή κάθε operation επεξεργάζεται batches αυτό σημαίνει ότι όλα τα operations στο Spark streaming οριοθετούνται χρονικά από το μέγεθος των batches ή από το μέγεθος των windows στην περίπτωση των window operations.

Το **Spark streaming** δουλεύει σε processing time. Αυτό σημαίνει ότι ο χρόνος έναρξης του πρώτου batch ή window (σε περίπτωση που έχουμε window-based operation) ταυτίζεται με το χρόνο έναρξης του Spark application. Ο χρόνος έναρξης του Spark application ορίζεται από το system clock του machine .Το επόμενο batch ή window(αν έχουμε window-based operation) εκκινεί με τη λήξη του interval του πρώτου batch ή window (αν έχουμε window-based operation). Αρα συμπεραίνουμε ότι η πρόοδος του χρόνου εξαρτάται από το system clock του machine στο οποίο τρέχει το εκάστοτε application.

Η επεξεργασία ενός batch ή window (αν μιλάμε για window-based operation) εκκινεί όταν τελειώσει η επεξεργασία του προηγούμενου batch ή window (όταν μιλάμε για window-based operation). Για αυτό το λόγο ο χρόνος που ξεκινάει η επεξεργασία ενός batch ή window(αν μιλάμε για window-based operation) συνήθως δεν ταυτίζεται με τον χρόνο εκκίνησης του .

6.10 Flink: Αντίληψη του χρόνου

Το Flink λειτουργεί σε processing,event και ingestion time.Σε όλες τις περιπτώσεις χρησιμοποιεί timestamps.

Timestamps

Το Flink λειτουργεί με **timestamps**.Είναι ο χρόνος στον οποίο έχει δημιουργηθεί ένα record(event) ή ο χρόνος στον οποίο ένα record(event) μπαίνει μέσα στο Flink. Στην πρώτη περίπτωση το timestamp ανατίθεται από τον producing sensor ή το producing service και στη δεύτερη περίπτωση ανατίθεται από το Flink. Με τη χρήση των timestamps το Flink εξασφαλίζει ότι τα records διαβάζονται με χρονική σειρά.

Το Flink όπως είπαμε λειτουργεί σε processing,event και ingestion time. Αυτοί οι τρόποι λειτουργίας καθορίζουν τον τρόπο με τον οποίο το Flink διαχειρίζεται τα records που δέχεται αλλά και τον τρόπο με τον οποίο το Flink αντιλαμβάνεται το χρόνο.

Αντίληψη του χρόνου σε σχέση με τα records

Στο **processing time** το Flink διαβάζει τα records χωρίς να λαμβάνει υπόψη του τα timestamps που περιέχει το κάθε record.Με άλλα λόγια δέχεται όλα τα εισερχόμενα records ανεξάρτητα από τα timestamps που περιέχουν.Στο **event time** το Flink , διαβάζει records, εξάγει από κάθε record το timestamp που περιέχει και του έχει ανατεθεί από το producing device κατά τη στιγμή της δημιουργίας του και φροντίζει τα records να διαβάζονται με χρονική σειρά. Όσα records μπαίνουν μέσα στο Flink και έχουν timestamps που παραβιάζουν την χρονική σειρά , απορρίπτονται καθώς θεωρούνται out-of-order και δεν συμμετέχουν στην διαμόρφωση του output. Στο **ingestion time** το Flink διαβάζει records και σε κάθε record αναθέτει ένα timestamp που έχει την τιμή του χρόνου στον οποίο το record μπαίνει στο Flink . Και σε αυτή την περίπτωση το Flink φροντίζει ώστε τα records να διαβάζονται με χρονική σειρά.Όσα records έχουν timestamps που παραβιάζουν την χρονική σειρά , απορρίπτονται καθώς θεωρούνται out-of-order και δεν συμμετέχουν στην διαμόρφωση του output.

Αντίληψη του χρόνου σε σχέση με τα operations

Τα operations στο Flink (όταν μιλάμε για streaming) χωρίζονται σε δύο κατηγορίες. Στα operations με windows και στα operations που δεν έχουν windows.

Στην περίπτωση των **non-windowed** operations , η έναρξη του κάθε operation καθορίζεται από το πότε θα έρθουν τα πρώτα στοιχεία στο Flink είτε έχουμε processing είτε event είτε ingestion time.

Στην περίπτωση των **windowed operations** ,όταν το Flink δουλεύει σε **processing time** ,ο χρόνος έναρξης του πρώτου window ταυτίζεται με το χρόνο έναρξης του Flink application. Ο χρόνος έναρξης του Flink application καθορίζεται από το system clock του machine .Το επόμενο window εκκινεί με τη λήξη του interval του πρώτου window . Άρα συμπεραίνουμε ότι η πρόοδος του χρόνου εξαρτάται από το system clock. Όταν το Flink δουλεύει σε event ή σε ingestion time,η έναρξη του πρώτου window γίνεται όταν τα πρώτα στοιχεία μπαίνουν μέσα στο window.Το επόμενο window εκκινεί με τη λήξη του interval του πρώτου window . Άρα συμπεραίνουμε ότι η πρόοδος του χρόνου εξαρτάται από το δεδομένα.

6.11 Ορισμός backpressure

Backpressure αναφέρεται στην κατάσταση όπου ένα processing engine λαμβάνει δεδομένα σε υψηλότερο ρυθμό από αυτόν που μπορεί να επεξεργαστεί λόγω μιας προσωρινής αιχμής φορτίου(load spike) όπως φαίνεται και στην παρακάτω εικόνα.

Fig. 22: Backpressure

Αν το backpressure δεν αντιμετωπιστεί σωστά μπορεί στη χειρότερη περίπτωση να οδηγήσει σε απώλεια στοιχείων. Σε αυτήν την περίπτωση ο buffer του processing engine είναι γεμάτος με αποτελεσμα όσα καινούρια records έρχονται να τα πετάει μέχρις ότου να αδειάσει μέρος του buffer (από records που έχουν πάει για process) προκειμένου να μπορέσει να δεχτεί τα επόμενα εισερχόμενα records. Αυτή η περίπτωση φαίνεται στο παρακάτω σχημα.

Fig. 23: Οριακή περίπτωση του Backpressure

Στο flink κινούμαστε με απώλεια στοιχείων σε σχέση με το backpressure ενώ στο spark streaming με απώλεια στοιχείων όσο και με αυξανόμενη ούρα στα batches που περιμενουν για processing.

6.12 Sustainable throughput

Στα πειράματα το Spark και το Flink δέχονται τα input streams με παραλληλία 1 . Για κάθε stream που δέχεται ένας operator ως input διεξάγουμε δέκα διαφορετικές μετρήσεις. Ξεκινάμε με το inputstream να έχει 10 kafka partitions και φτάνουμε τα 100 partitions με βήμα 10 partitions . Αποκλείουμε μετρήσεις στις οποίες έχουμε την εμφάνιση **backpressure** στο Spark και στο Flink .Βρίσκουμε σε ποιον αριθμό partitions το Spark και το Flink έχει το μικρότερο **latency** για τον εκάστοτε operator .Το **latency** αναφέρεται στο χρόνο που κάνει το κάθε processing engine να καταγράψει το συνολικό output του operation (μέχρι να γράψει και το τελευταίο στοιχείο) ξεκινώντας από τη στιγμή που ο kafka consumer (Spark ή Flink application) αρχίζει να διαβάζει το πρώτο στοιχείο του input stream.

Ο κατάλληλος συνδυασμός batch size και linger.ms μας βοήθαει να βρούμε τον καλύτερο χρόνο που σημειώνει το Spark και το Flink σε κάθε πείραμα.

Στα **διαγράμματα** που ακολουθούν σε κάθε πείραμα συγκρίνουμε το performance του Spark και του Flink σε σχέση με το μέσο sustainable throughput . Αυτός ο δείκτης υπολογίζεται ως εξής:

$$\textbf{μέσο sustainable throughput} = \frac{\text{αριθμός στοιχείων } \textit{input stream}}{\text{μικρότερο } \textit{latency}}$$

Όπου ο αριθμός στοιχείων inputstream στην περίπτωση του window join είναι το άθροισμα των στοιχείων των δύο streams που χρησιμοποιούνται ως input .

Στα πλαίσια του γενικότερου optimization των πειραμάτων χρησιμοποιήσαμε **snappy** compression για αύξηση του throughput.

6.13 Σύγκριση Spark και Flink σε σχέση με το backpressure

Στο παρακάτω διάγραμμα συγκρίνουμε το **latency** μεταξύ του Spark και του Flink μεταβάλλοντας τον αριθμό των kafka partitions των input streams για τον join operator . Το setup του πειράματος είναι 5 nodes , 16 cores.

Κόκκινο: Flink
Μπλε : Spark

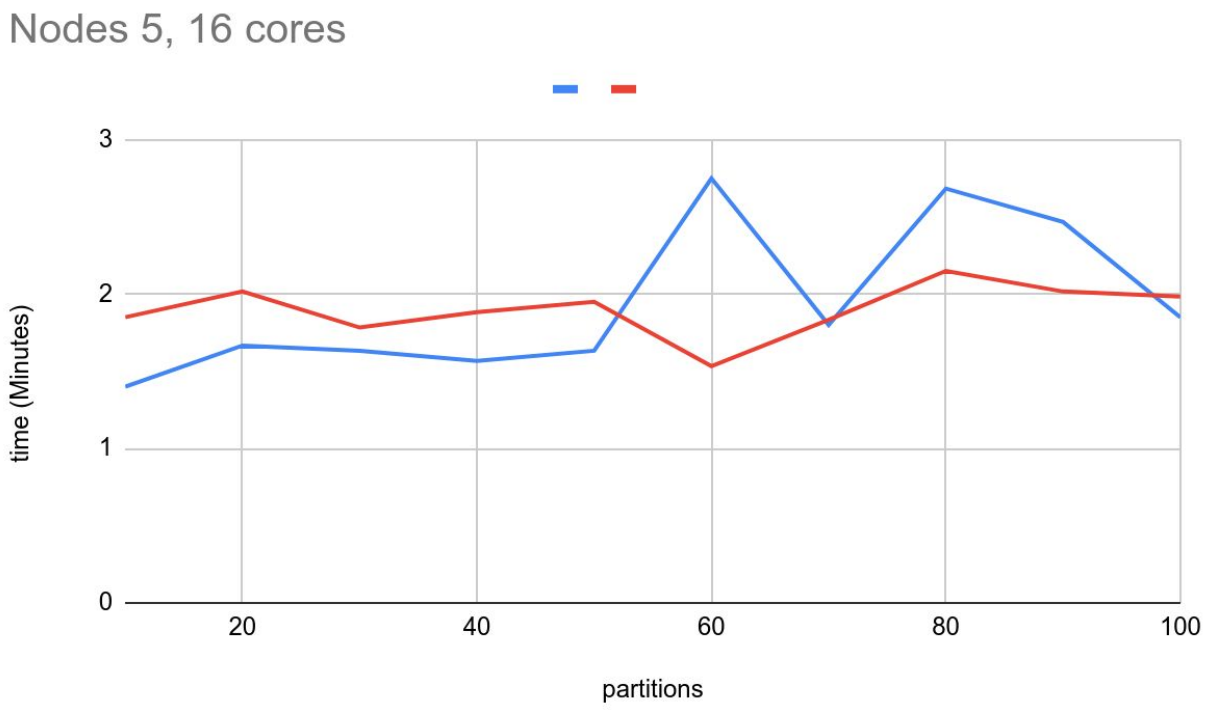


Fig. 24: Latency for windowed join in both processing engines depending on the number of kafka partitions of the input streams .

Όπως προαναφέραμε αυτή τη διαδικασία την ακολουθήσαμε σε κάθε πείραμα για την εύρεση του μέσου sustainable throughput.

Παρατηρούμε από το παραπάνω διάγραμμα ότι το Spark και το Flink έχουν fluctuations στο latency σε σχέση με τα partitions. Αυτό οφείλεται στο **backpressure**. Η απώλεια στοιχείων από ένα processing engine είναι η χειρότερη εκδοχή του backpressure. Πριν από την απώλεια στοιχείων το backpressure εμφανίζεται ως αύξηση του latency καθώς περιμένουν τα στοιχεία στους buffers για επεξεργασία λόγω του αυξημένου throughput που δεν μπορεί αφομοιώσει το processing engine.
Fluctuation στο latency (συναρτήσει των kafka partitions) έχουμε κατά βάση στο join και όχι τόσο στις υπόλοιπες συναρτήσεις(filter, window aggregate by key). Αυτό οφείλεται στο γεγονός ότι το window join είναι πιο ακριβό operation.
Το **Spark** όπως παρατηρούμε έχει περισσότερο fluctuation σε σχέση με το Flink καθώς είναι πιο επιρρεπές στο **backpressure** σε σχέση με το Flink . Αυτό οφείλεται στο τρόπο λειτουργίας του Spark . Το Spark εκτελεί πολλά jobs στο ίδιο batch interval.Μέχρι να εκτελεστεί το κάθε job το input rate περιορίζεται ενώ ο Dag scheduler συντονίζει και δρομολογεί όλα τα jobs. Αυτό σε συνδυασμό με το ότι το Spark περιμένει να τελειώσει το processing ενός batch για να πάει στο επόμενο κάνουν το Spark πιο επιρρεπές στο backpressure .

6.14 Χειρισμός kafka records από Spark και Flink

6.14.1 Spark Streaming

Όπως έχουμε αναφέρει ο Kafka λειτουργεί ως server ο οποίος αποθηκεύει stream records σε κατηγορίες που ονομάζονται topics. Ο Kafka οργανώνει τα records σε partitions και τα διατηρεί σε ένα κατανεμημένο log. Σε κάθε record που ανήκει σε κάποια partition του ανατίθεται ένα sequential id που ονομάζεται offset. Οι kafka consumers είναι υπεύθυνοι για την ανίχνευση της θέσης ενός record μέσα στο log δηλαδή του offset (Η θέση ενός record ονομάζεται και offset). Κάθε record που καταναλώνει ένας kafka consumer απαρτίζεται από ένα **κλειδί** , μια **τιμή** (value) και ένα **timestamp**.

Το Spark streaming όπως προαναφέραμε δουλεύει σε processing time. Δηλαδή κάθε batch με batch interval 10s για παράδειγμα περιλαμβάνει όλα τα records τα οποία φτάνουν στο operator του Spark μεταξύ των χρόνων που το system clock υποδεικνύει ως το συγκεκριμένο batch interval. Όταν το Spark δουλεύει με kafka consumers κάθε batch διαβάζει records από όλα τα partitions του kafka stream τα οποία έχουν φτάσει στον operator μέσα στο εκάστοτε batch interval και ανήκουν σε ένα συγκεκριμένο εύρος offset, όπως φαίνεται και στην παρακάτω εικόνα.

(Το Spark streaming δουλεύει με τα offsets των records που διαβάζει, δεν αναγνωρίζει τα timestamps που εσωκλείουν τα kafka records)

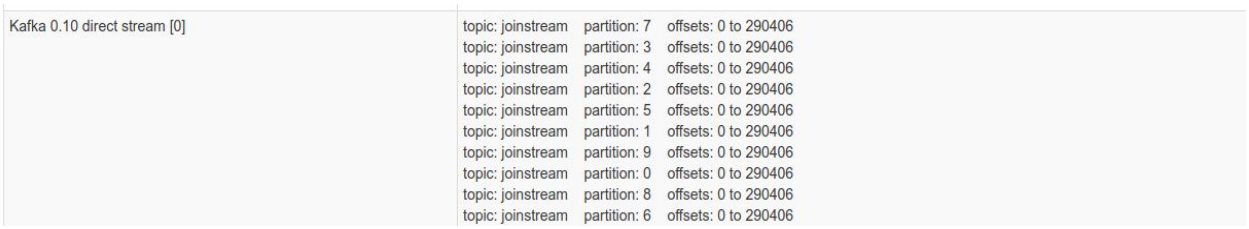


Fig. 25: Κατανομή των offsets των kafka records του inputstream που διαβάζει το batch κατά τη διάρκεια ενός operation στο Spark Streaming.

Το αμέσως επόμενο batch διαβάζει records από όλα τα partitions του kafka stream σε ένα συγκεκριμένο εύρος offset ξεκινώντας από το σημείο του offset που σταμάτησε να διαβάζει το προηγούμενο batch για κάθε partition όπως φαίνεται και στην παρακάτω εικόνα . Έτσι συνεχίζεται για όλα τα επόμενα batches μέχρι να τελειώσει το διάβασμα του input stream.

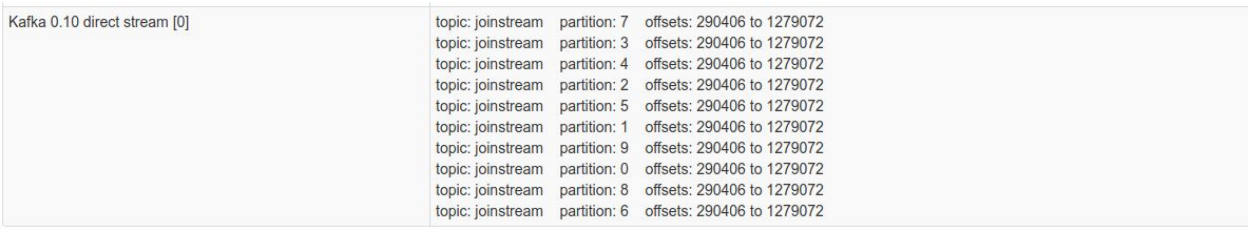


Fig. 26: Κατανομή των offsets των kafka records του inputstream που διαβάζει το αμέσως επόμενο batch κατά τη διάρκεια ενός operation στο Spark Streaming.

Στο spark streaming τα στοιχεία επεξεργάζονται ανα batch που σημαίνει ότι όσα στοιχεία μπαίνουν στο ίδιο batch επεξεργάζονται αυτόνομα από τα υπόλοιπα batches. Αν για παράδειγμα έχουμε ένα filter operation τότε το πρώτο batch (που διαβάζει τα records με offsets από 0 μέχρι 290406) υπολογίζει το output του operation ανεξάρτητα από το δεύτερο batch (που διαβάζει τα records με offsets από 29046 μέχρι 1279072)

6.14.2 Flink

Το Flink δουλεύει με **timestamps**. Συγκεκριμένα εξάγει από τα records που διαβάζει τα timestamps που τους έχουν αναθέσει ο kafka producer . Αυτά τα timestamps αναφέρονται σε event time δηλαδή έχουν ανατεθεί στα records τη στιγμή που τα παραλαμβάνει ο kafka producer και τα στέλνει στον kafka broker (server).
Όλα τα πειράματα μας σε σχέση με το Flink τρέχουν σε **processing time**. Έτσι όλα τα time-based operations όπως τα windows χρησιμοποιούν το system clock των machines που τρέχουν το εκάστοτε operation. Έτσι,ο χρόνος έναρξης του πρώτου window ταυτίζεται με το χρόνο έναρξης του Flink application. Ο χρόνος έναρξης του Flink application καθορίζεται από το system clock του machine .Το επόμενο window εκκινεί με τη λήξη του interval του πρώτου window . Άρα συμπεραίνουμε ότι η πρόοδος του χρόνου εξαρτάται από το system clock.

Event time

Δεν χρησιμοποιήσαμε **event time processing** στους operators του **Flink** καθώς έχουμε απώλεια output. Το event time processing δεν εξαρτάται από κάποιο system clock. Αντιθέτως ο τρόπος για να σημειωθεί πρόοδος στο event time processing στο Flink είναι τα watermarks . Τα **watermarks** λειτουργούν ως μέρος του stream δεδομένων που βρίσκεται μέσα στο Flink και κουβαλάνε ένα timestamp t. Ένα watermark(t) δηλώνει ότι το event time έχει φτάσει στο χρόνο t σε ένα stream εννοώντας ότι όσα στοιχεία μπαίνουν μέσα στο Flink με timestamp t'<=t απορρίπτονται καθώς θεωρούνται out-of-order . Στη περίπτωση μας διαβάζουμε διαδοχικά διαφορετικά partitions του stream που στέλνει ο kafka. Κάθε partition έχει αναθέσει από ένα timestamp σε κάθε record του κατά τη δημιουργία του στον kafka.

Όπως φαίνεται στην παρακάτω εικόνα το Flink διαβάζει το ένα partition μετά το άλλο , αναθέτει ένα watermark στο stream και δεν το ανανεωνει παρά μόνο όταν μπαίνει ένα record με timestamp > τρέχοντος watermark.

Στην περίπτωση μας έχουμε μεγάλη απώλεια output καθώς πολλά από τα records που μπαίνουν στο Flink δεν πληρούν την προϋπόθεση timestamp>watermark. Αυτό συμβαίνει διότι το Flink διαβάζει με τυχαία σειρά τα partitions του Kafka. Το κάθε kafka partition έχει ascending timestamps για τα records του ωστόσο η σχέση των timestamps για records που ανήκουν σε διαφορετικά partitions είναι εντελώς τυχαία . Το watermark που έχει εξαχθεί από την εισαγωγή του πρώτου partition στο Flink απορρίπτει αρκετά records επόμενων partitions που μπαίνουν στο Flink και δεν πληρούν την προϋπόθεση timestamp>watermark καθώς τα θεωρεί out of order.

Η λειτουργία του Flink σε **event time** φαίνεται στην παρακάτω εικόνα.Τα νούμερα στα partitions της εικόνας είναι τα timestamps των records που τους έχουν ανατεθεί κατά τη δημιουργία τους στον kafka.

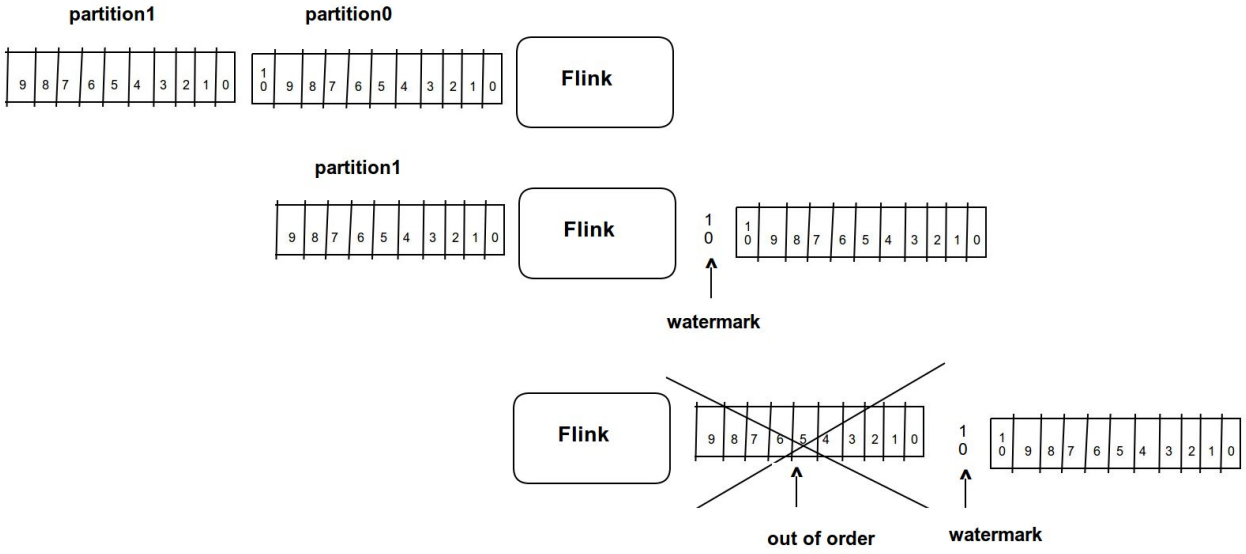


Fig. 27: Flink event-time operation

Ingestion time

Στο **ingestion time** , σε κάθε record που μπαίνει μέσα στο Flink ανατίθεται ένα timestamp με τιμή το current time του source operator . Με αυτό τον τρόπο όλα τα time-based operations αναφέρονται σε αυτό το timestamp. Το Flink διαβάζει το ένα kafka partition μετά το άλλο , αναθέτει ένα watermark στο stream και δεν το ανανεωνει παρά μόνο όταν μπαίνει κάποιο record που του ανατίθεται timestamp > τρέχοντος watermark. Στην περίπτωση του ingestion time δεν έχουμε απώλεια output καθώς το watermark που ανατίθεται στο stream δεν βρίσκει κανένα επόμενο record που μπαίνει στο Flink να έχει timestamp< watermark . Αυτό συμβαίνει διότι κάθε record που μπαίνει μέσα στο Flink παίρνει ένα timestamp με τιμή το current time του source operator . Δηλαδή κάθε record που μπαίνει στο Flink έχει το μεγαλύτερο timestamp σε σχέση με όλα τα υπόλοιπα records που έχουν ήδη μπει στο Flink . Συνεπώς κάθε record που μπαίνει έχει timestamp > watermark . Με αυτό τον τρόπο δεν απορρίπτεται κανένα record που μπαίνει μέσα στο Flink ως out of order.

Σε **ingestion time** λειτουργεί το πείραμα μας ωστόσο δεν χρησιμοποιήσαμε αυτή την περίπτωση καθώς το ingestion time δεν έχει απώλεια στοιχείων μόνο όταν το Flink διαβάζει ένα stream (Source(1)). Στη γενική περίπτωση που το Source έχει παραλληλία >1 έχουμε απώλεια στοιχείων.

ΚΕΦΑΛΑΙΟ 7:Filter

7.1 Περιγραφή

Στο filter βάζουμε ως input ένα stream που αποτελείται από 150.000.000 records.Οι τιμές των records ξεκινούν από το 1 και φτάνουν τα 150.000.000. Κάθε record μέσα στο stream έχει μοναδική τιμή. Το κάθε record είναι ένας **integer**. Το μέγεθος του κάθε record είναι μικρότερο από 10 bytes (κυμαίνεται από 1 μέχρι 9 bytes ανάλογα με την τιμή του record).

Το **filter** γενικά φιλτράρει records που πληρούν μια συγκεκριμένη συνθήκη.Για το filter διεξήγαμε τρία πειράματα . Και στα τρία πειράματα πειράξαμε τις μεταβλητές **linger.ms** και **batch size** του Kafka producer για μεγιστοποίηση του throughput με το οποίο στέλνονται τα records στον Kafka broker και από εκεί καταναλώνονται από τα application του Spark και του Flink , τα οποία λειτουργούν ως Kafka consumers. Επίσης όπως προαναφέραμε βρήκαμε τον αριθμό των kafka partitions του inputstream για τον οποίο το Spark και το Flink (filter) application έχουν το μικρότερο latency σε σχέση με την παράμετρο μεταβολής για το κάθε πείραμα.

Στο **πρώτο πείραμα** κρατάμε σταθερό το inputstream και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster. Σε αυτό το πείραμα το filter έχει **selectivity 0.5** καθώς ο αριθμός των στοιχείων που επιλέγονται είναι ο μισός από τον αριθμό των στοιχείων του input stream. Σε αυτό το selectivity επιλέγονται records που πληρούν τη συνθήκη: **record mod 2==0.** Δηλαδή επιλέγονται records που διαιρούνται με το 2.

Κρατώντας σταθερό το input stream τρέχουμε το application του Spark και του Flink για το filter στα εξής set-ups. 5 nodes-16 cores , 4 nodes-8 cores, 4 nodes-4 cores . Σε όλα τα set-up έχουμε 4GB RAM για το driver του Spark και τον Job Manager του Flink αντίστοιχα και 4 GB RAM για τους workers του Spark και τους Task Managers του Flink αντίστοιχα.

Στο **δεύτερο πείραμα** κρατάμε πάλι σταθερό το inputstream και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster αλλά αυτή τη φορά το filter έχει **selectivity 0.33**.Σε αυτό το selectivity επιλέγονται records που πληρούν τη συνθήκη: **record mod 3==0.** Δηλαδή επιλέγονται records που διαιρούνται με το 3.

Στο **τρίτο πείραμα** διατηρούμε σταθερό το setup του cluster (5 nodes-16 cores, 4GB driver, 4GB worker) και εισάγουμε διαδοχικά τρεις φορές το ίδιο input stream. Κάθε φορά διαφοροποιείται το filter condition και για αυτό το λόγο μεταβάλλεται το **selectivity.**

7.2 ΔΙΑΓΡΑΜΜΑΤΑ

Τα διαγράμματα είναι χωρισμένα με βάση την παράμετρο μεταβολής.

Στα **διαγράμματα** που ακολουθούν συγκρίνουμε το Spark και το Flink για κάθε πείραμα που διεξάγουμε σε σχέση με το **μέσο sustainable throughput** που αναφέραμε σε προηγούμενο κεφάλαιο.Το throughput μετριέται σε **records/sec** .

Για τον operator του filter η μεταβολή του cluster έχει ως εξής : 4 nodes-4 cores
4 nodes-8 cores
5 nodes-16 cores

7.2.1 CORES

1ο πείραμα

Σε αυτό το πείραμα βάζουμε ένα **inputstream** που αποτελείται από 150.000.000 records. Το **output** του filter operation είναι 50.000.000 records άρα έχουμε selectivity 0.33.

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το batch size για το application του Flink είναι 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 110. Για το application του Spark το batch size είναι 1.000.000 bytes και ο αριθμός των partitions του inputstream είναι 30.

Στα **8 cores** το batch size για το application του Flink είναι 4.000.000 bytes και ο αριθμός των partitions του inputstream είναι 100. Για το application του Spark 3.000.000 bytes και ο αριθμός των partitions του inputstream είναι 60.

Στα **4 cores** το batch size για το application του Flink είναι 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 20. Για το application του Spark 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 40.

Latency

Στα **16 cores** το latency για το application του Flink είναι 2,15 .Για το application του Spark το latency είναι 2,14 .

Στα **8 cores** το latency για το application του Flink είναι 2,17 .Για το application του Spark το latency είναι 2,31 .

Στα **4 cores** το latency για το application του Flink είναι 2,15 .Για το application του Spark το latency είναι 2,34.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

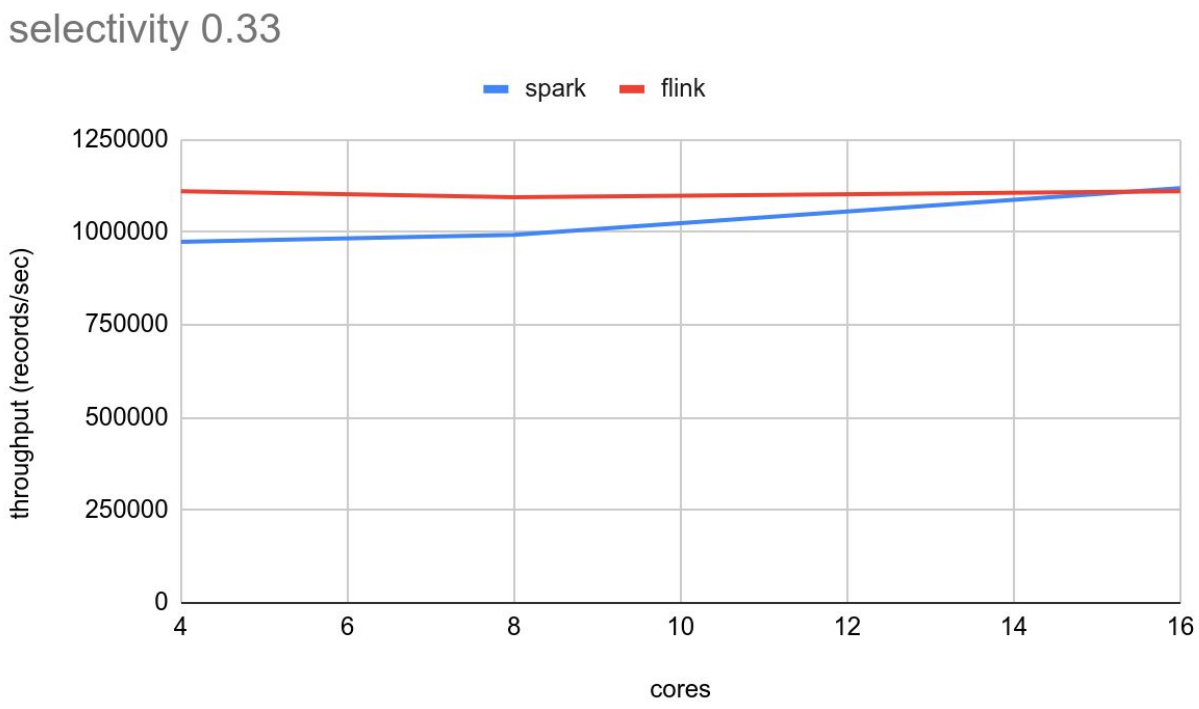


Fig. 28: Sustainable throughput for filter in both processing engines depending on the number of cores

2ο πείραμα

Σε αυτό το πείραμα βάζουμε το ίδιο **inputstream** με το προηγούμενο πείραμα. Το **output** του filter operation είναι 75.000.000 records άρα έχουμε selectivity 0.5.

Μεταβάλλουμε τα cores του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το batch size για το application του Flink είναι 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 20. Για το application του Spark το batch size είναι 3.000.000 bytes και ο αριθμός των partitions του inputstream είναι 100.

Στα **8 cores** το batch size για το application του Flink είναι 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 20. Για το application του Spark 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 40 .

Στα **4 cores** το batch size για το application του Flink είναι 3.000.000 bytes και ο αριθμός των partitions του inputstream είναι 20. Για το application του Spark 3.000.000 bytes και ο αριθμός των partitions του inputstream είναι 80 .

Latency

Στα **16 cores** το latency για το application του Flink είναι 2,33 .Για το application του Spark το latency είναι 2,43 .

Στα **8 cores** το latency για το application του Flink είναι 2,39 .Για το application του Spark το latency είναι 2,53 .

Στα **4 cores** το latency για το application του Flink είναι 2,19 .Για το application του Spark το latency είναι 2,38.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

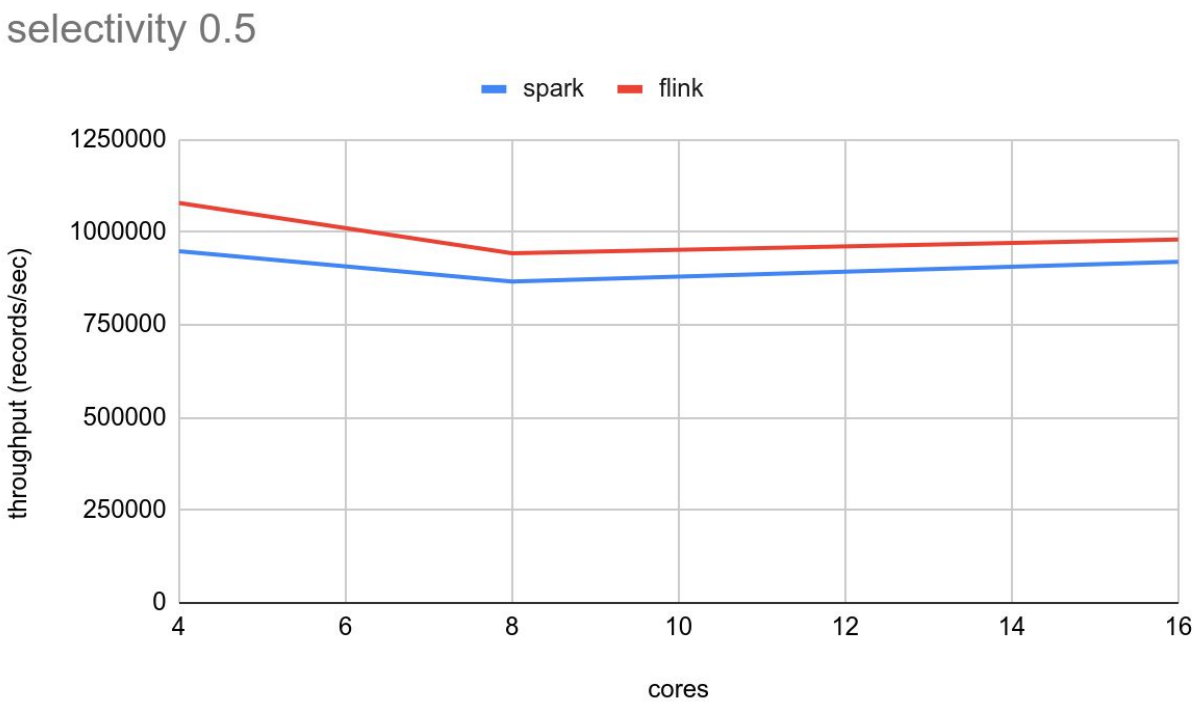


Fig. 29: Sustainable throughput for filter in both processing engines depending on the number of cores

Σχολιασμός

Παρατηρούμε ότι το Flink είναι καλύτερο από το Spark . Αυτό οφείλεται στο γεγονός ότι πρώτον το Spark έχει **blocking operators**. Αυτό σημαίνει ότι πρέπει να περιμένουμε να ολοκληρωθεί ένα στάδιο του operation για να πάμε στο επόμενο. ’Οπως φαίνεται και στην παρακάτω εικόνα πρέπει να περιμένουμε να υποστούν repartition όλα τα records του batch για να προχωρήσουμε στο filter.

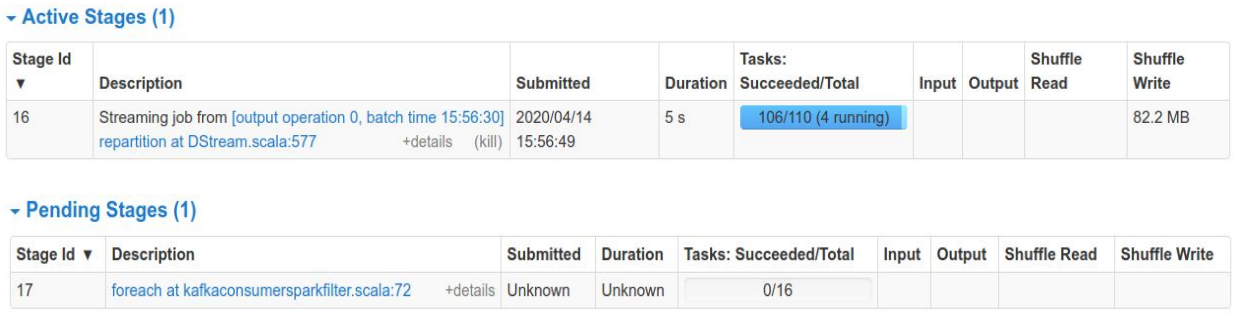


Fig. 30: Η σειρά εκτέλεσης των stages του filter στο Spark Streaming

Το pending stage είναι το stage του filtering και περιμένει να ολοκληρωθεί το active stage του repartition των records.

Δεύτερον στο Spark παράγονται **διαφορετικά είδη RDDs** στα διαφορετικά στάδια του application όπως φαίνεται και στην παρακάτω εικόνα που είναι το Spark graph για το filter.

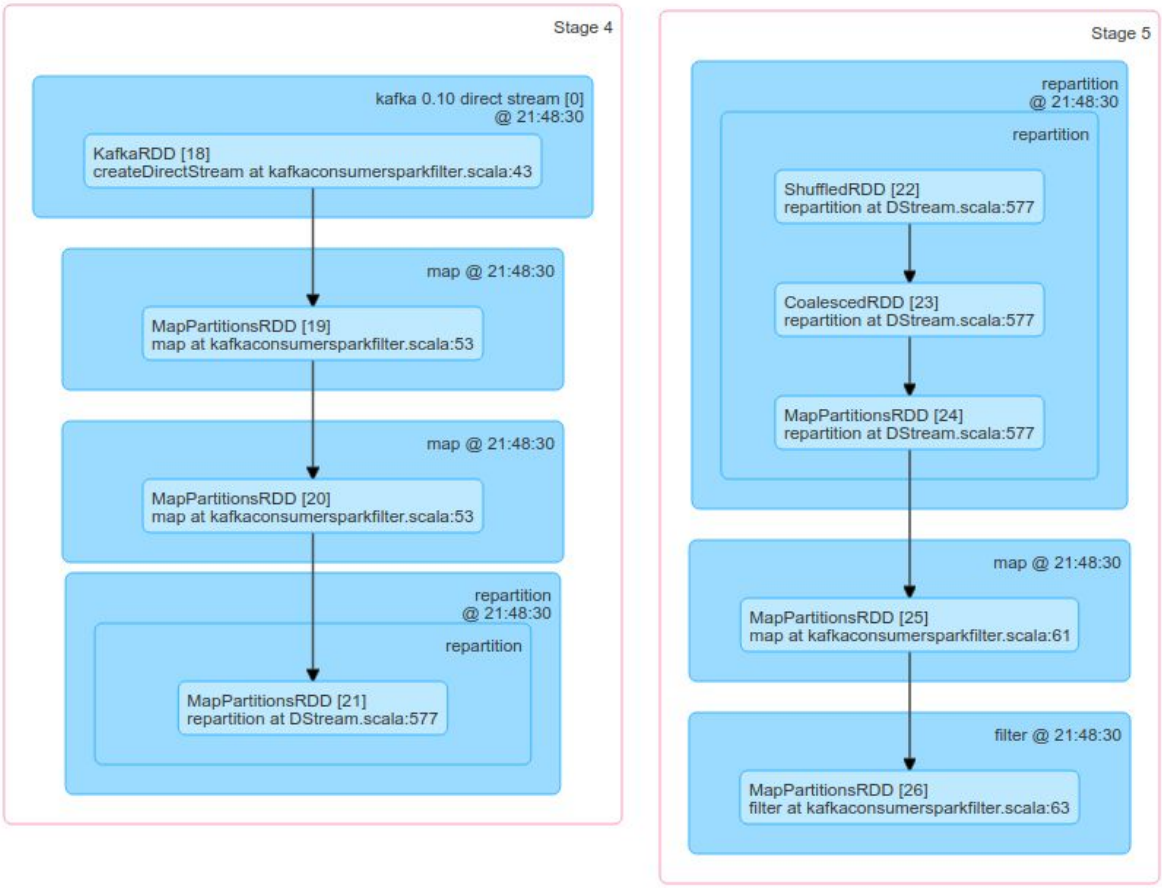


Fig. 31: DAG for filter operation

Το Flink αντίθετα δεν έχει **blocking operators**. Αντιθέτως το Flink προωθεί κάθε record που έρχεται διαδοχικά σε όλα τα stages του operation υπό την προϋπόθεση ότι έχει ολοκληρωθεί η επεξεργασία του προηγούμενου record. (chaining processing).
Τα στάδια ενός job λειτουργούν σαν αλυσίδα όπου κάθε στάδιο παίρνει επιτόπου records που έχει επεξεργαστεί το προηγούμενο στάδιο χωρίς να περιμένουμε να έχουν υποστεί επεξεργασία συγκεκριμένος αριθμός από records (batch).
Δεν παράγει ενδιάμεσα αποτελέσματα (όπως κάνει το Spark με τα ενδιάμεσα RDDs) , όπως φαίνεται και στο παρακάτω Flink graph .

Fig. 32: Flink Dataflow for filter operation

7.2.2 SELECTIVITY

3ο πείραμα

Σε αυτό το πείραμα έχουμε ένα σταθερό setup στον cluster. Ο cluster αποτελείται από 5 nodes και 16 cores.

Μεταβάλλουμε το **selectivity** του filter . Το input stream αποτελείται από 150.000.000 records. Από το 1ο input stream επιλέγεται το 5% των στοιχειών (selectivity 0.05) , από το 2ο inputstream το 33% των στοιχείων (selectivity 0.33) και από το 3ο input stream επιλέγεται το 50% των στοιχείων (selectivity 0.5) . Στο **selectivity 0.05** επιλέγονται στοιχεία που διαιρούνται με το 20 άρα πληρούν την συνθήκη : **record mod 20==0**

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με το selectivity του filter .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στο **selectivity 0.05** το batch size για το application του Flink είναι 1.000.000 bytes και ο αριθμός των partitions του inputstream είναι 50 . Για το application του Spark το batch size είναι 1.000.000 bytes και ο αριθμός των partitions του inputstream είναι 40.

Στα **selectivity 0.33** το batch size για το application του Flink είναι 2 .000.000 bytes και ο αριθμός των partitions του inputstream είναι 110 . Για το application του Spark 1.000.000 bytes και ο αριθμός των partitions του inputstream είναι 30 .

Στα **selectivity 0.5** το batch size για το application του Flink είναι 2.000.000 bytes και ο αριθμός των partitions του inputstream είναι 20. Για το application του Spark 3.000.000 bytes και ο αριθμός των partitions του inputstream είναι 100.

Latency

Στο **selectivity 0.05** το latency για το application του Flink είναι 2,13 .Για το application του Spark το latency είναι 2,08 .

Στα **selectivity 0.33** το latency για το application του Flink είναι 2.15 .Για το application του Spark το latency είναι 2.14.

Στα **selectivity 0.5** το latency για το application του Flink είναι 2.33 .Για το application του Spark το latency είναι 2.43 .

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή του selectivity του filter operator για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

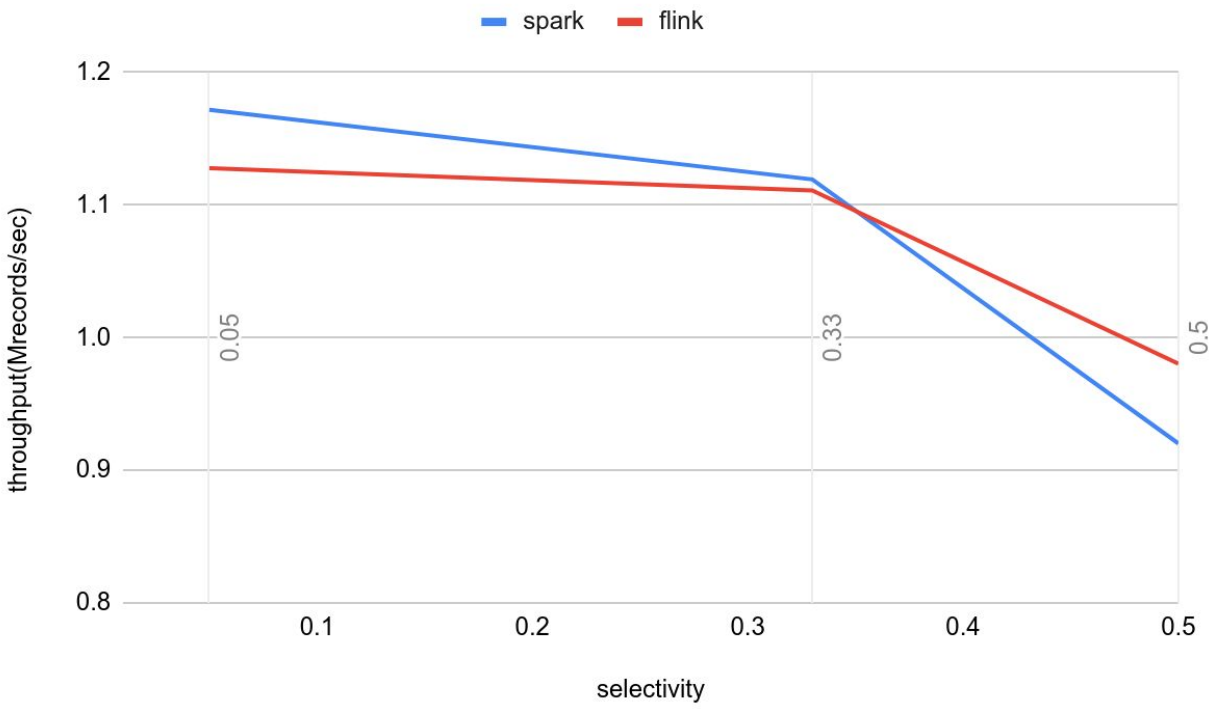


Fig. 33: Sustainable throughput for filter in both processing engines depending on the selectivity of the filter operator

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο στο selectivity 0.05 , οριακά καλύτερο στο selectivity 0,33 και χειρότερο στο selectivity 0,5.

Ο λόγος για την διαφορά στο performance μεταξύ του Spark και του Flink βρίσκεται στον limitation του network bandwidth του Flink .

Το setup είναι 5 nodes και 16 cores.Γνωρίζουμε ότι όσο αυξάνονται τα nodes το performance του Flink περιορίζεται από το network bandwidth. Στο filter δεν έχουμε reshuffling των data με βάση το key (hash) αφού το filter δεν είναι keyed operation. Άρα ο διάδρομος στο Flink χρησιμοποιείται μόνο στο rebalance όπως φαίνεται και στον παρακάτω Flink graph. Στο rebalance το inputstream έχει ίδιο αριθμό στοιχείων και στις τρεις περιπτώσεις selectivity και από τη στιγμή που έχουμε ίδιο setup και στα τρία selectivities ο περιορισμός του network bandwidth είναι κοινός. Ωστόσο το limitation αυτό δεν φαίνεται στο selectivity 0.5 . Αυτό συμβαίνει διότι στο 0.5 είναι αρκετά μεγαλύτερο το output οπότε ο όγκος των παραπάνω δεδομένων που πρέπει να γραφούν επισκιάζουν το limitation του network bandwidth. Στα

selectivities 0.5 και 0.33 φαίνεται το limitation γιατί μειώνεται το output και έτσι το Spark έχει καλύτερο performance από το Flink.

Fig. 34: Flink Dataflow for filter operation

ΚΕΦΑΛΑΙΟ 8: Aggregate by key

8.1 Περιγραφή

Το **aggregate by key** εφαρμόζεται σε streams που έχουν records της μορφής **(key, value)** και η λειτουργία του είναι να προσθέτει τις τιμές των records που έχουν κοινό κλειδί.

Στο aggregate by key βάζουμε ως **input** ένα stream που αποτελείται από 100.000.000 records. Το inputstream έχει records της μορφής **(value).** Κάθε record είναι ένας **integer** . Το μέγεθος του κάθε record είναι μικρότερο ή ίσο των 3 bytes(ανάλογα με την τιμή του record). Η τιμή του κάθε record ανήκει σε ένα διάστημα ακεραίων τιμών που ορίζουμε εμείς κάθε φορά.
Το αρχικό inputstream με records της μορφής **(value)** περνάει από έναν map operator όπου στο output κάθε record έχει μετατραπεί σε **(value, 1) .** Με αυτόν το τρόπο έχουμε μετατρέψει το αρχικό stream σε ένα stream με records της μορφής **(key,value)** και έτσι μπορεί να εφαρμοστεί το aggregate by key.

Στην περίπτωση μας έχουμε ένα **sliding window** 20s με sliding interval 10s. Άρα το aggregate by key εφαρμόζεται σε records που ανήκουν στο ίδιο window . Επειδή έχουμε sliding interval μικρότερο από το window size , τα windows αλληλοεπικαλύπτονται και έτσι το output του κάθε window υπολογίζεται με records που ανήκουν και σε προηγούμενο και σε επόμενο window.Το aggregate by key στην περίπτωση μας είναι time-based operation .

Διεξάγουμε τρία πειράματα. Και στα τρία πειράματα πειράξαμε τις μεταβλητές **linger.ms** και **batch size** του Kafka producer για μεγιστοποίηση του throughput με το οποίο στέλνονται τα records στον Kafka broker και από εκεί καταναλώνονται από τα application του Spark και του Flink , τα οποία λειτουργούν ως Kafka consumers. Επίσης όπως προαναφέραμε βρήκαμε τον αριθμό των kafka partitions του inputstream για τον οποίο το Spark και το Flink application έχουν το μικρότερο latency σε σχέση με την παράμετρο μεταβολής για το κάθε πείραμα.

Στο **πρώτο πείραμα** διατηρούμε σταθερό το setup του cluster (5 nodes-16 cores, 4GB driver, 4GB worker) και εισάγουμε διαδοχικά τέσσερα input streams με ίδιο αριθμό στοιχείων και **διαφορετικό αριθμό κλειδιών**.

Στο **δεύτερο πείραμα** κρατάμε σταθερό το inputstream και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster. Κρατώντας σταθερό το input stream τρέχουμε το application του Spark και του Flink για το aggregate by key στα εξής set-ups. 5 nodes-16 cores , 4 nodes-8

cores, 4 nodes-4 cores, 2 nodes-2 cores . Σε όλα τα set-up έχουμε 4GB RAM για το driver του Spark και τον Job Manager του Flink αντίστοιχα και 4 GB RAM για τους workers του Spark και τους Task Managers του Flink αντίστοιχα.

Στο **τρίτο πείραμα** κρατάμε σταθερό το inputstream και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster.Αυτή τη φορά το inputstream έχει **random κατανομή**.Κρατώντας σταθερό το input stream τρέχουμε το application του Spark και του Flink για το aggregate by key στα εξής set-ups. 5 nodes-16 cores , 4 nodes-8 cores, 4 nodes-4 cores, 2 nodes-2 cores . Σε όλα τα set-up έχουμε 4GB RAM για το driver του Spark και τον Job Manager του Flink αντίστοιχα και 4 GB RAM για τους workers του Spark και τους Task Managers του Flink αντίστοιχα.

8.2 Τι είναι το trigger στο Flink

Το **trigger** καθορίζει πότε πρέπει ένα window στο Flink να αρχίσει να υπολογίζει τα αποτελέσματα του window operation . Ένα trigger για παράδειγμα μπορεί να υπολογίζει το αποτέλεσμα του window operation όταν μπουν μέσα στο window 100 στοιχεία.

Στο **aggregate by key** έχουμε ένα window 20s και ένα sliding interval 10s .Το trigger για να υπολογιστεί το window aggregate γίνεται όταν μέσα στο window υπάρχουν records τα οποία έχουν ίδιο key και άρα μπορεί να γίνει aggregate by key .

8.3 ΔΙΑΓΡΑΜΜΑΤΑ

Για το aggregate by key σε κάθε πείραμα μεταβάλλεται μια παράμετρος. Τα διαγράμματα είναι χωρισμένα με βάση την παράμετρο μεταβολής.

Στα **διαγράμματα** που ακολουθούν συγκρίνουμε το Spark και το Flink για κάθε πείραμα που διεξάγουμε σε σχέση με το **μέσο sustainable throughput** που αναφέραμε σε προηγούμενο κεφάλαιο.Το throughput μετριέται σε **records/sec** .

Για τον operator του aggregate by key η μεταβολή του cluster έχει ως εξής : 2 nodes-2 cores
4 nodes-4 cores
4 nodes-8 cores
5 nodes-16 cores

8.3.1 Number of keys

1ο πείραμα

Σε αυτό το πείραμα έχουμε ένα σταθερό setup στον cluster. Ο cluster αποτελείται από 5 nodes και 16 cores.

Όπως προαναφέραμε το inputstream στο aggregate by key έχει records της μορφής **(value)** και η τιμή του value παίρνει τιμές από ένα εύρος ακεραίων αριθμών που εμείς ορίζουμε . Μετά το map operation το stream έχει records της μορφής **(value,1)** όπου το value ειναι το κλειδι .

Μεταβάλλουμε τον **αριθμό των κλειδιών** του inputstream . Το πρώτο input stream έχει κλειδιά που παίρνουν τιμές από ένα εύρος 4 ακέραιων αριθμών, το δεύτερο input stream έχει 10 κλειδιά ,το τρίτο stream έχει 50 κλειδιά και το τέταρτο stream έχει 100 κλειδιά.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms και το **batch size** έχει τιμή 1.000.000 bytes για όλες τις μετρήσεις.
Στα **4 keys** για το application του Flink ο αριθμός των partitions του inputstream είναι 60. Για το application του Spark ο αριθμός των partitions του inputstream είναι 10.

Στα **10 keys** για το application του Flink ο αριθμός των partitions του inputstream είναι 90 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 30 .

Στα **50 keys** για το application του Flink ο αριθμός των partitions του inputstream είναι 70 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 30.

Στα **100 keys** για το application του Flink ο αριθμός των partitions του inputstream είναι 90. Για το application του Spark ο αριθμός των partitions του inputstream είναι 30.

Latency

Στα **4 keys** το latency για το application του Flink είναι 1,35 .Για το application του Spark το latency είναι 1,14 .

Στα **10 keys** το latency για το application του Flink είναι 1,35 .Για το application του Spark το latency είναι 1,14 .

Στα **50 keys** το latency για το application του Flink είναι 1,40 .Για το application του Spark το latency είναι 1,20.

Στα **100 keys** το latency για το application του Flink είναι 1,35 .Για το application του Spark το latency είναι 1,18.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των κλειδιών του inputstream για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

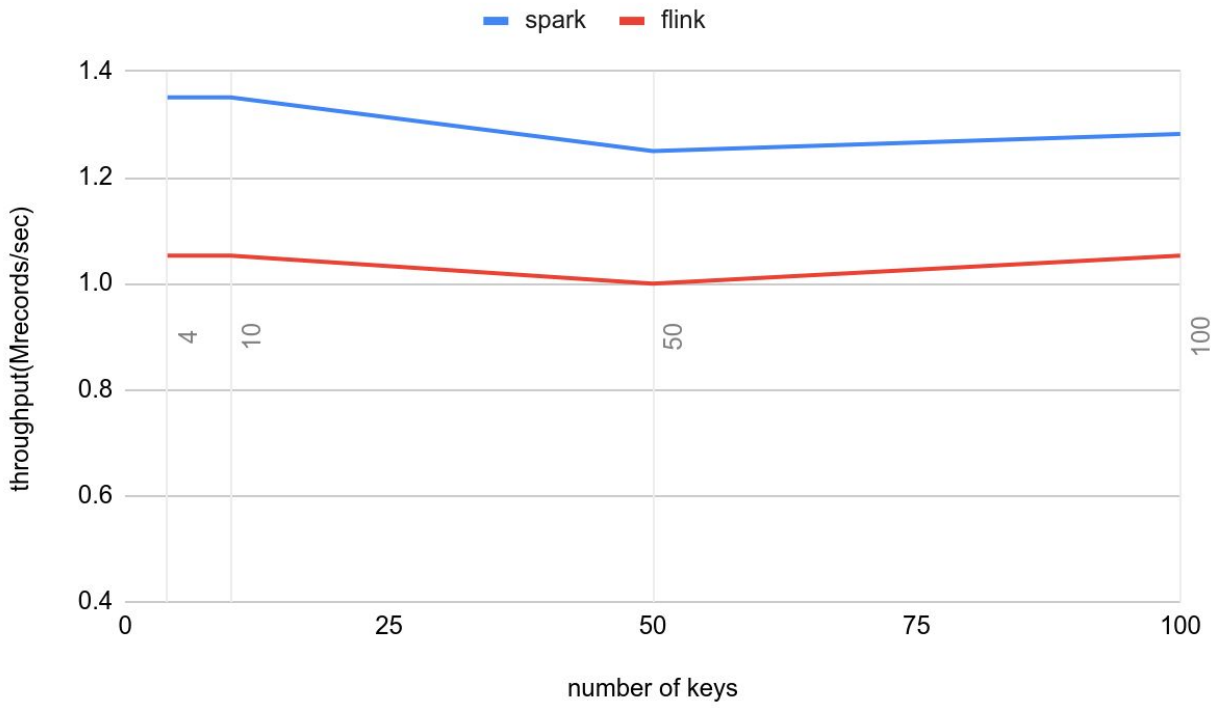


Fig. 35: Sustainable throughput for windowed aggregate by key in both processing engines depending on the number of keys .

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink .

Η μορφή του **1ου inputstream** είναι:

for i=1 to 100.000.000
Record = random. nextInt(4)

Δηλαδή το record παίρνει τιμή από 1 μέχρι 4 σε τυχαία κατανομή.

Στα υπόλοιπα κλειδιά η κατανομή είναι και αυτή random.

Περιγραφή Flink graph

Στο Flink τα δεδομένα αφού κατανέμονται στα partitions των nodes με round-robin τρόπο στην συνέχεια υφίστανται ανακατανομή στα partitions των nodes με βάση το κλειδί (hash) προτού γίνει το aggregate by key . Άρα το data shuffling είναι **εκτεταμένο**.

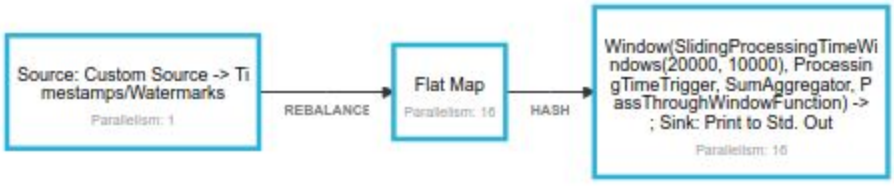


Fig. 36: Flink Dataflow for aggregate by key operation

Το data shuffling στο στάδιο του hash είναι εκτεταμένο λόγω του γεγονότος ότι οι κατανομές των streams είναι **random** (που σημαίνει ότι η πιθανότητα σε ένα partition να έχουμε πολλά records με το ίδιο key είναι μικρή ,προκειμένου να γίνει το aggregate by key) και λόγω του ότι το key έχει ένα **dimension**. Για δεδομένο μέγεθος input stream όσο μικρότερο είναι το dimension

του key τόσο αυξάνεται ο αριθμός των records με κοινό key. Αυτό σημαίνει ότι αυξάνεται το μέγεθος του aggregate by key operation και συνεπώς το data shuffling.

Tree aggregate -tree reduce

Το Spark υλοποιεί ένα tree aggregate tree reduce pattern το οποίο **ακολουθεί το mapreduce μοντέλο και ελαχιστοποιεί το communication και το data shuffling.**
Αρχικά όλα τα εισερχόμενα στοιχεία κατανέμονται στους executors . Στη συνέχεια οι executors εκτελούν map operation . Σε αυτό το operation εκτός του ότι μετατρέπουν τα records από (value) σε (value,1) προκειμένου να εκτελεστεί το aggregate by key σε επόμενο στάδιο, ανταλλάσουν μεταξύ τους **κάποια** records προκειμένου τα records του κάθε executor να ομαδοποιηθούν σε buckets με βάση το κλειδί. Με αυτό τον τρόπο **ελαχιστοποιείται το data shuffling** καθώς έχουμε ανταλλαγή λίγων στοιχείων μεταξύ των executors.
Στο τελευταίο στάδιο ο reducer(aggregate by key) κάνει fetch buckets του stream που έχουν κοινά κλειδιά για να εκτελέσει το aggregate by key . Τα parallel reduce tasks δεν επικοινωνούν μεταξύ τους καθώς κάθε task έχει τα σωστά στοιχεία για την εκτέλεση του aggregate by key. Με αυτό τον τρόπο ελαχιστοποιείται και το **communication**

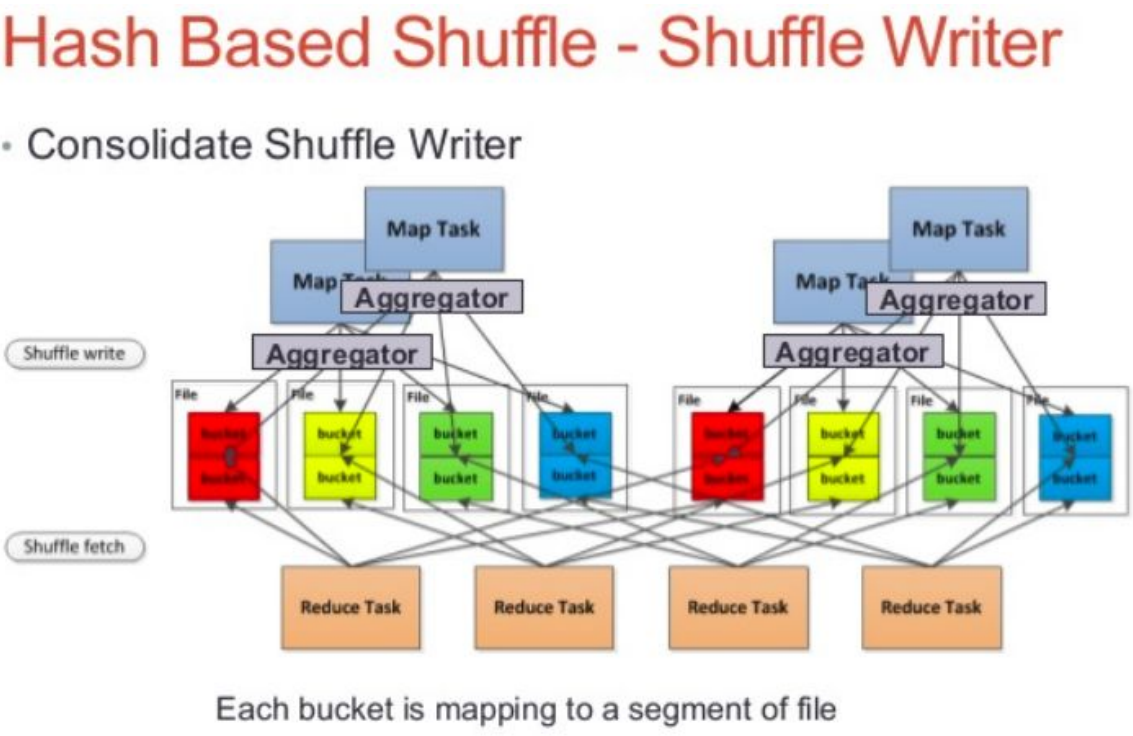


Fig. 37: Tree aggregate-tree reduce pattern

8.3.2 CORES

2ο πείραμα

input stream: 100.000.000 records
Output: (199, 50.000.000)
(200, 50.000.000)
(201, 50.000.000)
(202, 50.000.000)

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms και το **batch size** έχει τιμή 1.000.000 bytes για όλες τις μετρήσεις.

Στα **16 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 50. Για το application του Spark ο αριθμός των partitions του inputstream είναι 10.

Στα **8 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 50 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 10 .

Στα **4 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 20 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 10.

Στα **2 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 70 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 20.

Latency

Στα **16 cores** το latency για το application του Flink είναι 1,33 .Για το application του Spark το latency είναι 1,23 .

Στα **8 cores** το latency για το application του Flink είναι 1,32 .Για το application του Spark το latency είναι 1,25.

Στα **4 cores** το latency για το application του Flink είναι 1,40 .Για το application του Spark το latency είναι 1,24 .

Στα **2 cores** το latency για το application του Flink είναι 1,33 .Για το application του Spark το latency είναι 1,53 .

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

Κόκκινο: Flink
Μπλε : Spark

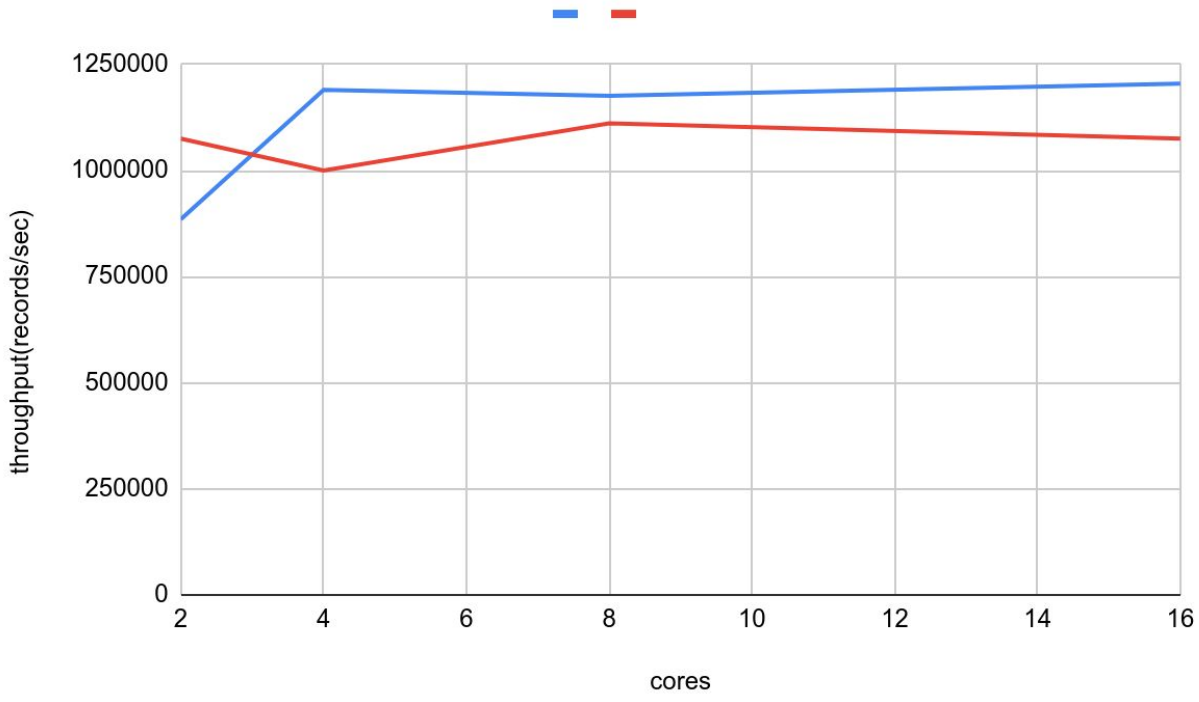


Fig. 38: Sustainable throughput for windowed aggregate by key in both processing engines depending on the number of cores.

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink . Αυτό οφείλεται στο **data skew**.

Η μορφή του **inputstream** είναι:

```
For i =199 to 201
      For j=1 to 25.000.000
            Record =i
```

Το **Flink** μετά το αρχικό rebalance που τα στοιχεία χωρίζονται με round-robin τρόπο στα partitions,στη συνέχεια ανακατανείμει τα στοιχεία ανάλογα με το key (hashing) για να προχωρήσει στο aggregate by key όπως φαίνεται στην παρακάτω εικόνα.

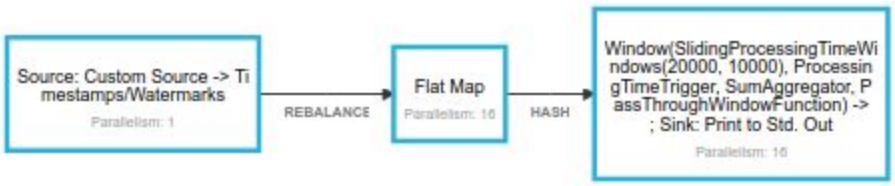


Fig. 39: Flink Dataflow for aggregate by key operation

Στο στάδιο του hashing το Flink χάνει πολύ χρόνο καθώς λόγω της δομής που έχει το stream ,στο hashing πρέπει να σταλούν όλα τα στοιχεία των partitions σε ένα partition για να γίνει το aggregate. Αυτό συμβαίνει γιατί τα στοιχεία του stream έχουν ίδιο key και το key αυτό αλλάζει αφότου έχουν διαβαστεί 25.000.000 στοιχεία. Για αυτό το λόγο το input stream θεωρείται **skewed**. Όταν μπαίνει το καινούριο key έχει ήδη υπολογιστεί το aggregate για το προηγούμενο key.

Λόγοι για τη διαφορά στο performance

Ένας λόγος για την διαφορά στο performance μεταξύ του Spark και του Flink βρίσκεται στον τρόπο με τον οποίο το Flink πραγματοποιεί aggregations .Το Flink χρησιμοποιεί ένα task slot για κάθε operator instance. Όταν τα data είναι skewed (που σημαίνει ότι χρειάζονται ανακατανομή) το Flink σε πολλά παράλληλα instances του aggregate (κάθε instance υλοποιείται σε ένα slot) υλοποιεί ανακατανομή δεδομένων προκειμένου να έρθουν τα σωστά(αυτά που έχουν κοινά keys) και να γίνει το aggregate by key.
Το Spark αντίθετα υιοθετεί ένα tree aggregate και tree reduce communication pattern για να ελαχιστοποιήσει το communication και το data shuffling.

Τέλος παρατηρούμε ότι στο setup των 2 nodes το Flink έχει καλύτερο performance από το Spark. Το Spark δουλεύει με blocking operators. Όσο μειώνονται τα nodes του cluster , αυτός ο περιορισμός γίνεται κυρίαρχος παράγοντας στη διαμόρφωση του latency.

3ο πείραμα

input stream: 100.000.000 records
Output:(0,20008996)
(1,19989244)
(2,19996056)
(3,20003930)
(4,19991746)
(5,19999004)
(6,20012146)
(7,19994890)
(8,19991110)
(9,20012878)

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms και το **batch size** έχει τιμή 1.000.000 bytes σχεδόν για όλες τις μετρήσεις.

Στα **16 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 50. Για το application του Spark ο αριθμός των partitions του inputstream είναι 40.

Στα **8 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 100 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 10 .

Στα **4 cores** για το application του Flink ο αριθμός των partitions του inputstream είναι 70 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 30.

Στα **2 cores** για το application του Flink το batch size είναι 2.000.000 bytes, ο αριθμός των partitions του inputstream είναι 80 . Για το application του Spark ο αριθμός των partitions του inputstream είναι 20.

Latency

Στα **16 cores** το latency για το application του Flink είναι 1,34 .Για το application του Spark το latency είναι 1,16.

Στα **8 cores** το latency για το application του Flink είναι 1,34 .Για το application του Spark το latency είναι 1,17.

Στα **4 cores** το latency για το application του Flink είναι 1,33 .Για το application του Spark το latency είναι 1,18 .

Στα **2 cores** το latency για το application του Flink είναι 1,34 .Για το application του Spark το latency είναι 2,02 .

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

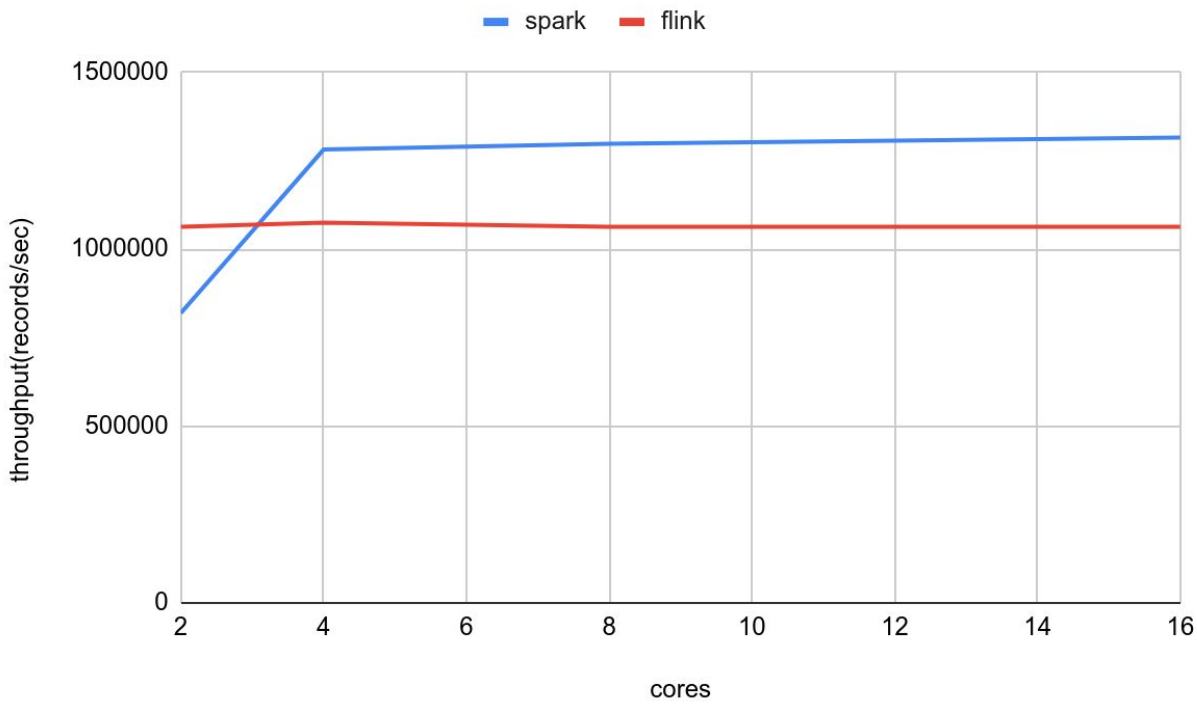


Fig. 40: Sustainable throughput for windowed aggregate by key in both processing engines depending on the number of cores.

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink .

Η μορφή του **inputstream** είναι:

for i=1 to 100.000.000
Record = random. nextInt(10)

Δηλαδή το record παίρνει τιμή από 1 μέχρι 10 σε τυχαία κατανομή. Έχει δέκα κλειδιά.

Στο **Flink** τα δεδομένα αφού κατανέμονται στα partitions των nodes με round-robin τρόπο στην συνέχεια υφίστανται ανακατανομή στα partitions των nodes με βάση το κλειδί (hash) προτού γίνει το aggregate by key . Άρα το data shuffling είναι **εκτεταμένο**.

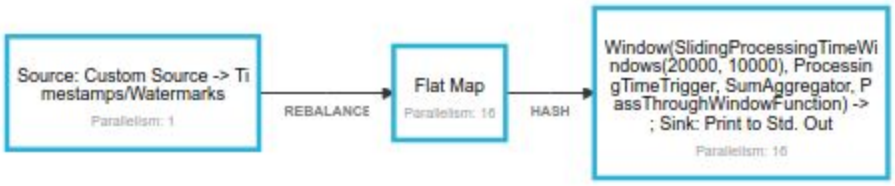


Fig. 41: Flink Dataflow for aggregate by key operation

Το data shuffling στο στάδιο του hash είναι εκτεταμένο και λόγω του γεγονότος ότι η κατανομή του stream είναι **random** (που σημαίνει ότι η πιθανότητα σε ένα partition να έχουμε πολλά records με το ίδιο key είναι μικρή ,προκειμένου να γίνει το aggregate by key) και στο γεγονός ότι το key έχει ένα **dimension**. Για δεδομένο μέγεθος input stream όσο μικρότερο είναι το dimension του key τόσο αυξάνεται ο αριθμός των records με κοινό key. Αυτό σημαίνει ότι αυξάνεται το μέγεθος του aggregate by key operation και συνεπώς το data shuffling.

Το Spark όπως προαναφέραμε υλοποιεί ένα tree aggregate tree reduce pattern το οποίο μειώνει κατά πολύ τον όγκο των δεδομένων που πρέπει να μεταφερθούν μέσω του δικτύου .

Τέλος παρατηρούμε ότι στο setup των 2 nodes το Flink έχει καλύτερο performance από το Spark. Το Spark δουλεύει με blocking operators. Όσο μειώνονται τα nodes του cluster , αυτός ο περιορισμός γίνεται κυρίαρχος παράγοντας στη διαμόρφωση του latency.

ΚΕΦΑΛΑΙΟ 9: Window join

9.1 Περιγραφή

Στο **window join** παίρνουμε δύο περιπτώσεις. Σε κάθε περίπτωση διεξάγουμε τρία πειράματα.

Η **πρώτη περίπτωση** περιλαμβάνει input streams με **one-dimension** key.Η **δεύτερη περίπτωση** περιλαμβάνει input streams με two-dimension keys. Το join δέχεται ως input **δύο streams**.

Η πρώτη περίπτωση με τη δεύτερη δεν είναι συγκρίσιμες καθώς αν σε ένα πείραμα μεταβάλλουμε μόνο το dimension του key των συνολικών input streams (άθροισμα των δύο input streams που δέχεται το join ως input) θα πρέπει να διατηρηθεί σταθερός ο αριθμός των στοιχείων και το selectivity. Στη δική μας περίπτωση τα συνολικά input streams με διαφορετικό dimension στο key ,έχουν ίδιο selectivity αλλά διαφορετικό αριθμό στοιχείων.

9.2 Πρώτη περίπτωση

Η **πρώτη περίπτωση** περιλαμβάνει input streams με two-dimension keys. Στη πρώτη περίπτωση έχουμε ένα **tumbling window** διάρκειας 2.30 λεπτών.

Η **πρώτη περίπτωση** περιλαμβάνει input streams με **two-dimension** keys. Έχουμε ένα **tumbling window** διάρκειας 2.30 λεπτών και δύο input streams με records της μορφής **((key1,key2),V)** και **((key1,key2), W)** αντίστοιχα .Επιστρέφεται ως output του join ένα stream με records της μορφής **((key1,key2), (V,W))** όπου για το κάθε key(**(key1,key2)**) έχουμε όλους τους δυνατούς συνδυασμούς **(V,W)** .

Στα input streams το **key1** είναι ένας Integer, το **key2** επίσης ένας Integer και το **value** ένα String. Το μέγεθος του κάθε record είναι **μικρότερο ή ίσο από 40 bytes.**
Το join εφαρμόζεται πάνω σε records των δύο input streams που ανήκουν στο ίδιο window. Τα records ανήκουν μόνο σε ένα window καθώς το window είναι tumbling και έτσι δεν αλληλοεπικαλύπτονται τα windows.

Διεξάγουμε τρία πειράματα. Και στα τρία πειράματα πειράξαμε τις μεταβλητές **linger.ms** και **batch size** του Kafka producer για μεγιστοποίηση του throughput με το οποίο στέλνονται τα records στον Kafka broker και από εκεί καταναλώνονται από τα application του Spark και του Flink , τα οποία λειτουργούν ως Kafka consumers. Επίσης όπως προαναφέραμε βρήκαμε τον αριθμό των kafka partitions των δύο inputstreams για τον οποίο το Spark και το Flink στο join application έχουν το μικρότερο latency σε σχέση με την παράμετρο μεταβολής για το κάθε πείραμα.

Στο **πρώτο πείραμα** κρατάμε σταθερά τα δύο input streams με **selectivity 0.8** και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster. Όπως προαναφέραμε ,όταν λέμε ότι έχουμε selectivity 0.8 εννοούμε ότι από τον συνολικό αριθμό των records των δύο streams επιλέγεται το 80% αυτών των records για να εφαρμοστεί πάνω τους το join . Κρατώντας σταθερά τα δύο input streams τρέχουμε το application του Spark και του Flink για το window join στα εξής set-ups. 5 nodes-16 cores , 4 nodes-8 cores, 4 nodes-4 cores . Σε όλα τα set-up έχουμε 8GB RAM για το driver του Spark και τον Job Manager του Flink αντίστοιχα και 6 GB RAM για τους workers του Spark και τους Task Managers του Flink αντίστοιχα.

Στο **δεύτερο πείραμα** κρατάμε πάλι σταθερά τα δύο input streams και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster αλλά αυτή τη φορά το συνολικό input stream (άθροισμα των δύο input streams) έχει **selectivity 0.6**.

Στο **τρίτο πείραμα** διατηρούμε σταθερό το setup του cluster (5 nodes-16 cores, 8GB driver, 6GB worker) και εισάγουμε διαδοχικά τρία συνολικά input streams (άθροισμα των δύο input streams που δέχεται το join) με ίδιο αριθμό στοιχείων και **διαφορετικό selectivity**.Στο join, record θεωρείται το **tuple**.

9.3 Δεύτερη περίπτωση

Η **δεύτερη περίπτωση** περιλαμβάνει input streams με **one-dimension** key. Έχουμε ένα **tumbling window** διάρκειας 2 λεπτών και δύο input streams με records της μορφής **(key,V)** και **(key, W)** αντίστοιχα .Επιστρέφεται ως output του join ένα stream με records της μορφής **(key, (V,W))** όπου για το κάθε key έχουμε όλους τους δυνατούς συνδυασμούς **(V,W)** .

Στα input streams σε κάθε record το **key** είναι ένας Integer και το value ένα **String**. Το μέγεθος του κάθε record είναι μικρότερο ή ίσο από 38 bytes**.**
Το join εφαρμόζεται πάνω σε records των δύο input streams που ανήκουν στο ίδιο window. Τα records ανήκουν μόνο σε ένα window καθώς το window είναι tumbling και έτσι δεν αλληλοεπικαλύπτονται τα windows.

Διεξάγουμε τέσσερα πειράματα. Και στα τέσσερα πειράματα πειράξαμε τις μεταβλητές **linger.ms** και **batch size** του Kafka producer για μεγιστοποίηση του throughput με το οποίο στέλνονται τα records στον Kafka broker και από εκεί καταναλώνονται από τα application του Spark και του Flink , τα οποία λειτουργούν ως Kafka consumers. Επίσης όπως προαναφέραμε βρήκαμε τον αριθμό των kafka partitions των δύο inputstreams για τον οποίο το Spark και το Flink στο join application έχουν το μικρότερο latency σε σχέση με την παράμετρο μεταβολής για το κάθε πείραμα.

Στο **πρώτο πείραμα** κρατάμε σταθερά τα δύο inputstreams με **selectivity 0.8** και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster. Όταν λέμε ότι έχουμε selectivity 0.8 εννοούμε ότι ο λόγος του αθροίσματος των records του πρώτου stream και των records του δεύτερου stream που έχουν μεταξύ τους κοινά κλειδιά (τα records του πρώτου με τα records του δεύτερου) προς τον συνολικό αριθμό των records των δύο input streams είναι 0.8.
Κρατώντας σταθερά τα δύο input streams τρέχουμε το application του Spark και του Flink για το window join στα εξής set-ups. 5 nodes-16 cores , 4 nodes-8 cores, 4 nodes-4 cores . Σε όλα τα set-up έχουμε 8GB RAM για το driver του Spark και τον Job Manager του Flink αντίστοιχα και 6 GB RAM για τους workers του Spark και τους Task Managers του Flink αντίστοιχα.

Στο **δεύτερο πείραμα** κρατάμε πάλι σταθερά τα δύο input streams και μεταβάλλουμε τον αριθμό των cores και των nodes του cluster αλλά αυτή τη φορά το συνολικό input stream(άθροισμα των δύο input streams) έχει **selectivity 0.6**.

Στο **τρίτο πείραμα** διατηρούμε σταθερό το setup του cluster (5 nodes-16 cores, 8GB driver, 6GB worker) και εισάγουμε διαδοχικά τρία συνολικά input streams (άθροισμα των δύο input streams που δέχεται το join) με ίδιο αριθμό στοιχείων και **διαφορετικό selectivity**.

Στο **τέταρτο πείραμα** διατηρούμε σταθερό το setup του cluster (5 nodes-16 cores, 8GB driver, 6GB worker) και εισάγουμε διαδοχικά τρία συνολικά input streams (άθροισμα των δύο input streams που δέχεται το join) με ίδιο αριθμό στοιχείων και **διαφορετικό selectivity**. Τα συνολικά input streams που εισάγουμε έχουν **random κατανομή**.

9.4 Επιλογή window size προκειμένου να μην υπάρχει απώλεια output στο Flink

Αρχικά κάναμε πειράματα στο join με **window size 10s** και στην περίπτωση όπου τα input streams έχουν one-dimension key και στην περίπτωση όπου τα input streams έχουν two-dimensional key. Ωστόσο παρατηρήσαμε ότι και στις δύο περιπτώσεις το Flink είχε

μικρότερο output σε σχέση με το output στο Spark Streaming. Είχαμε δηλαδή απώλεια output στο Flink . Αυτό οφείλεται στον τρόπο που λειτουργεί το Flink σε σχέση με τα windowed-operations.

Το **Flink** συγκεκριμένα στο window join , κάνει join σε δύο streams που βρίσκονται μέσα στο ίδιο window. ΄Οταν αρχίζει ένα window μπαίνουν μέσα στοιχεία και από τα δύο streams μέχρι να τελειώσει το interval του τρέχοντος window. Τα επόμενα στοιχεία μπαίνουν σε επόμενο window. Το **join** ανάμεσα σε δύο streams **γίνεται για όσο χρόνο διαρκεί το window interval.** Όταν ξεκινήσει καινούριο window η διαδικασία εκκινεί από την αρχή και γίνεται join μεταξύ στοιχείων των δύο input streams που έχουν μπει στο δεύτερο window. Εάν τη στιγμή που ξεκινά το δεύτερο window δεν έχουν προλάβει να υπολογιστούν joins στοιχείων από το πρώτο window δεν θα υπολογιστούν ποτέ καθώς χρονικά έχουμε μπει στο επόμενο window. Δηλαδή τα transformations εκτελούνται **μόνο εντός του window interval .** Για αυτό στο window join έχουμε απώλεια output. Το join είναι βαρύ operation και χρειάζεται αρκετό χρόνο για να υπολογιστεί. Εάν μέσα στο window interval δεν έχουν προλάβει όλα τα στοιχεία του window με κοινά κλειδιά να γίνουν joined τότε έχουμε απώλεια output καθώς με τη λήξη του interval πάμε στο επόμενο window και σταματάει η επεξεργασία του προηγούμενου window .

Στο **Spark streaming** δεν έχουμε αυτό το πρόβλημα καθώς περιμένουμε να ολοκληρωθεί η επεξεργασία του κάθε window και μετά πάμε στην επεξεργασία του επόμενου window.Δηλαδή περιμένουμε να ολοκληρωθεί το join για το κάθε window και μετά πάμε στο επόμενο όπως φαίνεται και στην παρακάτω εικόνα. (Το batch size ταυτίζεται με το window size στην παρακάτω εικόνα)

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)
2020/07/17 20:27:37	752970 records	13 s	1 s	15 s
2020/07/17 20:27:36	62370 records	13 s	0.5 s	14 s
2020/07/17 20:27:35	50870 records	13 s	0.4 s	13 s
2020/07/17 20:27:34	84470 records	13 s	0.4 s	13 s
2020/07/17 20:27:33	76752 records	12 s	0.6 s	13 s
2020/07/17 20:27:32	85181 records	12 s	0.6 s	12 s
2020/07/17 20:27:31	80187 records	11 s	0.6 s	12 s
2020/07/17 20:27:30	406668 records	10 s	1 s	11 s
2020/07/17 20:27:29	250873 records	7 ms	10 s	10 s

Fig. 42: Κατανομή των records στα batches

Για να μην έχουμε απώλεια output στο window join στο Flink ,θα πρέπει τα δύο streams να διαβαστούν όσο νωρίτερα γίνεται από την εκπνοή του window interval ,έτσι ώστε όλο το join να έχει χρόνο να υπολογιστεί μέσα στο window interval προτού πάμε στο επόμενο window.

Σε όλα τα πειράματα του window join βάλαμε windows με interval μεγαλύτερο από το χρόνο που χρειάζεται για να μπουν τα στοιχεία μέσα στο window στο Flink έτσι ώστε το join να έχει χρόνο να υπολογιστεί μέσα στο window interval προτού πάμε στο επόμενο window και έτσι να μην έχουμε απώλεια output. Με αυτό το σκεπτικό και ανάλογα με τα input streams που χρησιμοποιούμε σε κάθε πείραμα καταλήξαμε στην περίπτωση των input streams με one-dimension key σε window 2 min και στην περίπτωση των input streams με two-dimensional keys σε window 2.30 min . Αυτές οι επιλογές εξασφαλίζουν ότι στο Flink δεν έχουμε απώλεια output και έτσι μπορούμε να συγκρίνουμε το performance του σε σχέση με το performance του Spark Streaming .

Στο window aggregate by key το Flink λειτουργεί με τον ίδιο τρόπο σε σχέση με τα windows. Ωστόσο δεν έχει απώλεια output καθώς το αποτέλεσμα του aggregate by key υπολογίζεται χρονικά εντός του interval του window που ανήκουν τα records (στα οποία εφαρμόζεται το aggregate by key) .Αυτό συμβαίνει διότι το aggregate by key υπολογίζεται γρήγορα και έτσι ο υπολογισμός του ανήκει χρονικά στο interval του window των records. Αντίθετα το join χρειάζεται περισσότερο χρόνο για να υπολογιστεί καθώς είναι πιο χρονοβόρο , πολύπλοκο και δαπανηρό σε σχέση με τα resources operation.

9.5 Batch interval ίσο με το window size στο Spark Streaming

Στο Spark streaming επιλέξαμε στα πειράματα για το window join και στις δύο περιπτώσεις ,το batch interval να είναι ίσο με το μέγεθος του window . Παρακάτω παραθέτουμε ένα παράδειγμα window join όπου στη πρώτη περίπτωση δοκιμάζουμε window 2 min με batch interval 1 min και στη δεύτερη περίπτωση δοκιμάζουμε window 2 min με batch interval 2 min.

Τα άθροισμα των στοιχείων των δύο inputs streams είναι 33.870.000 και το output του join θα πρέπει να είναι 93.225.000 tuples

Batch interval 1 min:

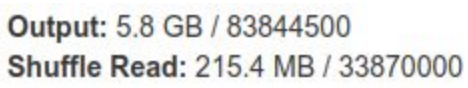
Το συνολικό inputstream των 33.870.000 στοιχείων χωρίζεται σε δύο batches όπως φαίνεται και στην παρακάτω εικόνα.

Completed Batches

2020/06/20 18:26:00	27783060 records	2 ms	53 s	53 s
2020/06/20 18:25:00	6086940 records	13 ms	0.3 s	0.3 s

Fig. 43: Κατανομή των δύο input streams στα batches

Το συνολικό output είναι :



Output: 5.8 GB / 83844500
Shuffle Read: 215.4 MB / 33870000

Παρατηρούμε ότι το output είναι μικρότερο από το επιθυμητό . Τα δύο input streams που βάλαμε ως είσοδο στο join έχουν την παρακάτω δομή

1ο stream:

```
for i=5001 to 3.039.000
        for j=1 to 5
                record =(i, “my site is spark examples”)
```

2ο stream:

```
for i=1 to 1.700.000
        for j=1 to 11
                record=(i, “my site is spark examples”)
```

Στο **πρώτο batch** θα κάνουμε process 1.298.700 tuples από το πρώτο stream και 4.788.240 tuples από το δεύτερο stream.

Άρα από το **1ο stream** θα έχουμε πάρει

```
For i=5001 to 264740
    For j=1 to 5
            record =(i, “my site is spark examples”)
```

Άρα από το **2ο stream** θα έχουμε πάρει

For i=1 to 435294
For j=1 to 11
record =(i, “my site is spark examples”)

For j =1 to 6
record =(435295, “my site is spark examples”)

Το **output** θα είναι : (264740-5001+1)*5*11=14.285.700 tuples

Στο **δεύτερο batch** θα κάνουμε process 13.871.300 tuples από το πρώτο stream και 13.911.760 tuples από το δεύτερο stream.

Άρα από το **1ο stream** θα έχουμε πάρει
For i=264741 to 3.039.000
For j=1 to 5
record =(i, “my site is spark examples”)

Άρα από το **2ο stream** θα έχουμε πάρει

For j=7 to 11
record =(435295, “my site is spark examples”)

For i=435296 to 1.700.000
For j=1 to 11
record =(i, “my site is spark examples”)

Το **output** θα είναι : (1700000-435296+1)*5*11+25=69.558.800 tuples

Το **συνολικό output** και των δύο batches είναι 83.844.500 tuples.

Στη συνέχεια δοκιμάσαμε το batch interval να είναι ίσο με το μέγεθος του window.

Batch interval 2 min:

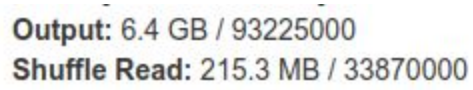
Το συνολικό inputstream των 33.870.000 στοιχείων ανήκει σε ένα batch όπως φαίνεται και στην παρακάτω εικόνα.

▾ Active Batches (1)

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Output Ops: Succeeded/Total
2020/06/19 21:20:00	33870000 records	10 ms	-	0/3 (3 running)

Fig. 44: Processing των input streams σε ένα batch

Το συνολικό output είναι :



Βλέπουμε ότι το window join με batch interval το μισό από το μέγεθος του window size δεν βγάζει σωστά αποτελέσματα .Αντιθέτως όταν το window size είναι ίσο με το batch interval τα αποτελέσματα είναι σωστά. Αυτό συμβαίνει διότι στην πρώτη περίπτωση το συνολικό inputstream (άθροισμα των δύο input streams) χωρίζεται σε δύο batches και έτσι χαλάει η δομή της εξάρτησης(εννοώντας τον αριθμό των στοιχείων που είναι κοινός μεταξύ των δύο streams) μεταξύ των δύο streams στα οποία γίνεται το join. Στο στάδιο του window αθροίζονται τα outputs των δύο batches όπως φαίνεται στον αριστερό γράφο της παραπάνω εικόνας. Το άθροισμα αυτό είναι μικρότερο από το επιθυμητό output.

Αντίθετα όταν το batch interval είναι ίσο με το window size τότε τα δύο input streams μπαίνουν σε ένα batch. Έτσι δεν χαλάει το selectivity και παράγεται το σωστό output του join το οποίο στη συνέχεια προωθείται στο window όπως φαίνεται στον δεξιό γράφο της εικόνας .

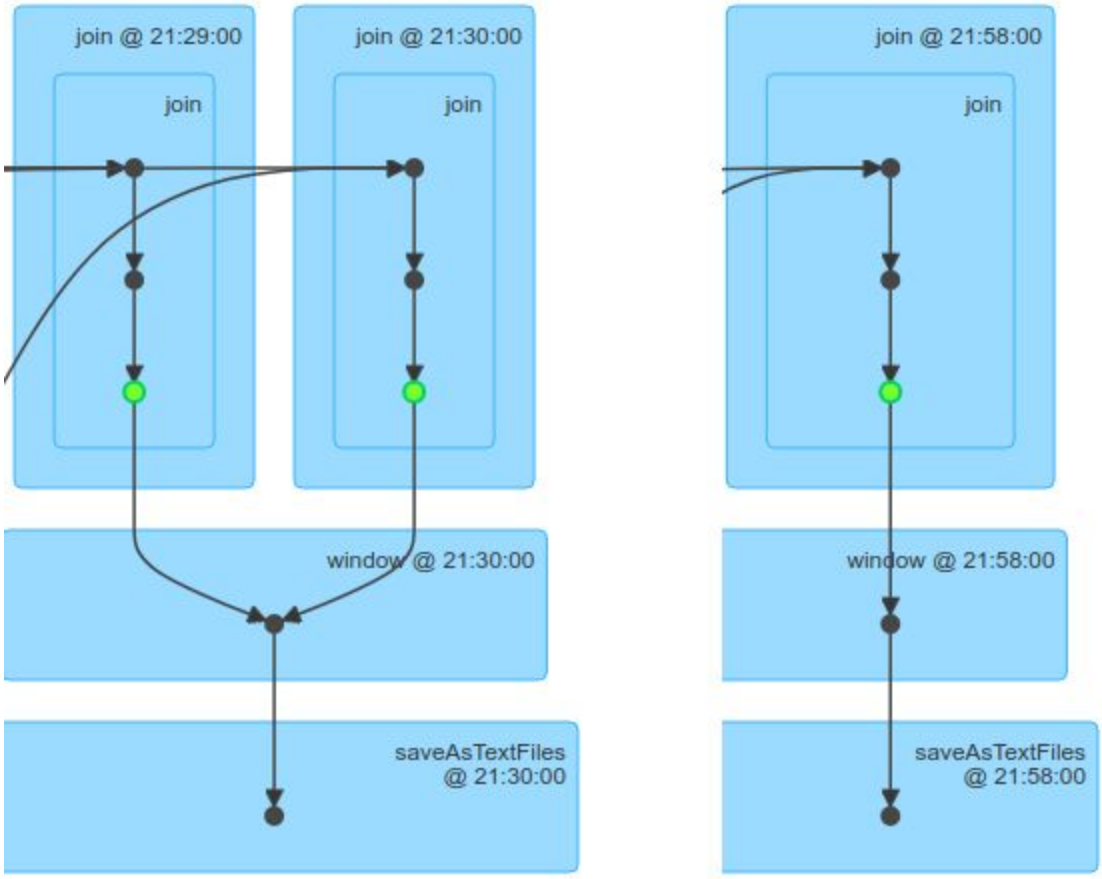


Fig. 45: Dag για την περίπτωση του window join με batch interval το μισό από το window interval(αριστερά), Dag για την περίπτωση που το batch interval είναι ίσο με το window interval (δεξιά).

9.6 Flink: Trigger στο join

Το trigger όπως προαναφέραμε καθορίζει πότε πρέπει ένα window στο Flink να αρχίσει να υπολογίζει τα αποτελέσματα του window operation . Ένα trigger για παράδειγμα μπορεί να υπολογίζει το αποτέλεσμα του window operation όταν μπουν μέσα στο window 100 στοιχεία.

Στο **window join** ο ορισμός του trigger είναι πιο περίπλοκος. Όπως φαίνεται στην παρακάτω εικόνα το κάθε source λαμβάνει ένα input stream . Το inputstream χωρίζεται στα Flink partitions στο rebalance. Όποιο record μπαίνει σε μια partition του ανατίθεται ένα id που είναι ίσο με την τιμή του record .Σε κάθε partition ανατίθεται ένα watermark που παίρνει τη μεγαλύτερη τιμή από όλα τα ids (για αυτό και το low watermark του ενός map operator εφτασε το 3.039.000 , πήρε την τιμή του μεγαλύτερου record,(for i=1 to 3.039.000)) **.** Το low watermark που παίρνει το map

operator είναι το μικρότερο από τα watermarks όλων των partitions του input stream. Το ίδιο συμβαίνει και στο δεύτερο source.

Fig. 46: Flink Dataflow for window join operation

Σε κάθε partition όταν μπει ένα record με τιμή μικρότερη από την τιμή του watermark , αυτό το record δεν απορρίπτεται ως out-of-order record (κάτι το οποίο θα συνέβαινε αν είχαμε event time). Αυτό συμβαίνει γιατί έχουμε processing time(η εξέλιξη του window operator εξαρτάται από το system clock και όχι από τα timestamps των records). Σε περίπτωση που μπει ένα record με τιμή μεγαλύτερη από την τιμή του watermark τότε το watermark παίρνει την τιμή του record. Ουσιαστικά τα watermarks σε αυτή την περίπτωση χρησιμοποιούνται ως δείκτες που σηματοδοτούν την έλευση νέων records.
Το operator του window join έχει ένα low watermark . Η τιμή του είναι ίση με τη μικρότερη τιμή μεταξύ των δύο low watermarks των map operators.

Το **low watermark** στο window λειτουργεί ως trigger. Το processing time window ξεκινάει να μετράει την αρχή του window από τη στιγμή που ξεκινάει να τρέχει το Flink application. Το trigger ουσιαστικά αυτό που κάνει είναι να λέει στο window πότε να υπολογίσει το output. Κάθε φορά που αλλάζει το low watermark του window τότε αρχίζει να υπολογίζεται το output του join από εκεί που το άφησε το join στο προηγούμενο watermark .Μπορεί το τρέχον join να συμπεριλάβει στοιχεία τα οποία έχουν έρθει στο window από προηγούμενα watermarks.

Το ότι το watermark του window έχει την τιμή του μικρότερου watermark από τους δύο map operators δεν έχει κάποια ιδιαίτερη σημασία στο processing time. Είναι σαν μια σύμβαση. Όλη αυτή η διαδικασία με τα watermarks γίνεται γιατί πρέπει να εξασφαλίσουμε triggers για το window . Πρέπει δηλαδή να ορίσουμε κάθε πότε το window θα υπολογίζει το join .
Το Flink δεν έχει blocking operators και άρα δεν υπάρχει κάποιο group στοιχειων χρονικά ορισμένο (batch) που να περιμένει να διαβαστεί για να ξεκινήσει μετά το στάδιο του join.

Το **Spark** δεν χρειάζεται triggers διότι εκκινεί το window όταν τελειώσει το διαβασμα των στοιχείων που ανήκουν στο ίδιο batch (όταν το batch size είναι ίσο με το window size). Στην

περίπτωση που το batch interval είναι μικρότερο από το window size τότε το window εκκινεί όταν φορτώσουν όλα τα batches που ανήκουν στο ίδιο window.

Όταν στο Flink βάζουμε μόνο ένα input stream στο join ως είσοδο , το window δεν έχει low watermark που σημαίνει ότι δεν κάνει ποτέ trigger (δεν υπολογίζει output) πράγμα λογικό από τη στιγμή που έχουμε ένα inputstream και άρα δεν μπορούμε να υπολογίσουμε join.

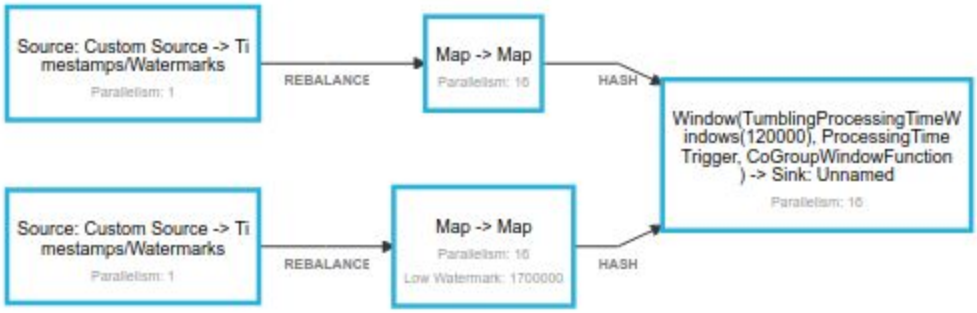


Fig. 47: Flink Dataflow for window join operation

Στο Spark streaming στο join κάθε batch διαβάζει records από τα δύο διαφορετικά streams που συνδυάζονται στο join. Αυτά τα records ανήκουν σε ένα kafka partition και κάθε partition έχει ένα συγκεκριμένο εύρος offset όπως φαίνεται στην παρακάτω εικόνα.

Kafka 0.10 direct stream [1]	topic: filterstream1 partition: 3 offsets: 2720000 to 5440000 topic: filterstream1 partition: 7 offsets: 2720000 to 5440000 topic: filterstream1 partition: 8 offsets: 2720000 to 5440000 topic: filterstream1 partition: 4 offsets: 2720000 to 5440000 topic: filterstream1 partition: 0 offsets: 2720000 to 5440000 topic: filterstream1 partition: 6 offsets: 2720000 to 5440000 topic: filterstream1 partition: 1 offsets: 2720000 to 5440000 topic: filterstream1 partition: 5 offsets: 2720000 to 5440000 topic: filterstream1 partition: 2 offsets: 2720000 to 5440000 topic: filterstream1 partition: 9 offsets: 2720000 to 5440000
Kafka 0.10 direct stream [0]	topic: filterstream partition: 3 offsets: 2427200 to 4854400 topic: filterstream partition: 7 offsets: 2427200 to 4854400 topic: filterstream partition: 4 offsets: 2427200 to 4854400 topic: filterstream partition: 8 offsets: 2427200 to 4854400 topic: filterstream partition: 9 offsets: 2427200 to 4854400 topic: filterstream partition: 5 offsets: 2427200 to 4854400 topic: filterstream partition: 1 offsets: 2427200 to 4854400 topic: filterstream partition: 6 offsets: 2427200 to 4854400 topic: filterstream partition: 0 offsets: 2427200 to 4854400 topic: filterstream partition: 2 offsets: 2427200 to 4854400

Fig. 48: Κατανομή των offsets των kafka records των δύο input streams που διαβάζει το batch κατά το join operation στο Spark Streaming.

9.7 Διαγράμματα

9.7.1 Πρώτη περίπτωση

[Join , tumbling window 2.30minutes,two- dimension key]

Στο join σε κάθε πείραμα μεταβάλλεται μια παράμετρος. Τα διαγράμματα είναι χωρισμένα με βάση την παράμετρο μεταβολής.

Στα **διαγράμματα** που ακολουθούν συγκρίνουμε το Spark και το Flink για κάθε πείραμα που διεξάγουμε σε σχέση με το **μέσο sustainable throughput** που αναφέραμε σε προηγούμενο κεφάλαιο.Το throughput μετριέται σε records/sec .

9.7.1.1 Cores

Για τον operator του join η μεταβολή του cluster έχει ως εξής : 4 nodes-4 cores
4 nodes-8 cores
5 nodes-16 cores

1ο πείραμα

Σε αυτό το πείραμα βάζουμε 2 input streams . Το **input stream1** αποτελείται από 24.272.000 tuples. Το **input stream 2** αποτελείται από 27.200.000 tuples.
Το **output** του join operation είναι 72.885.000 tuples και έχουμε selectivity 0.8.

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το batch size για το application του Flink είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 90. Για το application του Spark το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 20 .

Στα **8 cores** το batch size για το application του Flink είναι 7.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 20. Για το application του Spark το batch size είναι 7.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 20.

Στα **4 cores** το batch size για το application του Flink είναι 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 100. Για το application του Spark το batch size είναι 10.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 60.

Latency

Στα **16 cores** το latency για το application του Flink είναι 2,10 .Για το application του Spark το latency είναι 2,20.

Στα **8 cores** το latency για το application του Flink είναι 2,23 .Για το application του Spark το latency είναι 2,39.

Στα **4 cores** το latency για το application του Flink είναι 2,56 .Για το application του Spark το latency είναι 3,23.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

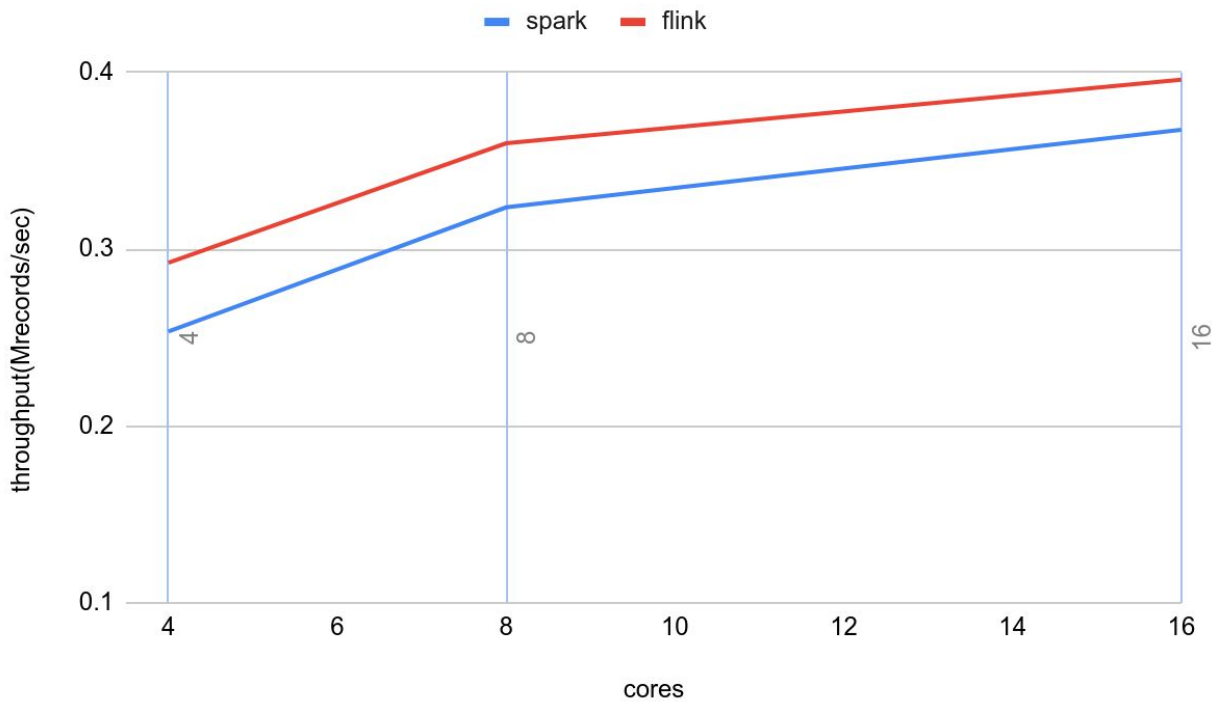


Fig. 49: Sustainable throughput for windowed join in both processing engines depending on the number of cores .

2ο πείραμα

Σε αυτό το πείραμα βάζουμε 2 input streams . Το **input stream1** αποτελείται από 24.272.000 tuples. Το **input stream 2** αποτελείται από 27.200.000 tuples.
Το **output** του join operation είναι 44.070.000 tuples και έχουμε selectivity 0.6.

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις του Spark . Το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το linger.ms για το application του Flink είναι 15.000 ms **,**το batch size είναι 250.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 80 . Για το application του Spark το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 20.

Στα **8 cores** το linger.ms για το application του Flink είναι 15.000 ms **,**το batch size είναι 450.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 100. Για το application του Spark το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 70.

Στα **4 cores** το linger.ms για το application του Flink είναι 1000 ms **,**το batch size είναι 13.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 90. Για το application του Spark το batch size είναι 9.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 40.

Latency

Στα **16 cores** το latency για το application του Flink είναι 1,55 .Για το application του Spark το latency είναι 1,59.

Στα **8 cores** το latency για το application του Flink είναι 2,20 .Για το application του Spark το latency είναι 2,29.

Στα **4 cores** το latency για το application του Flink είναι 3 .Για το application του Spark το latency είναι 3,18.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

Κόκκινο: Flink
Μπλε : Spark

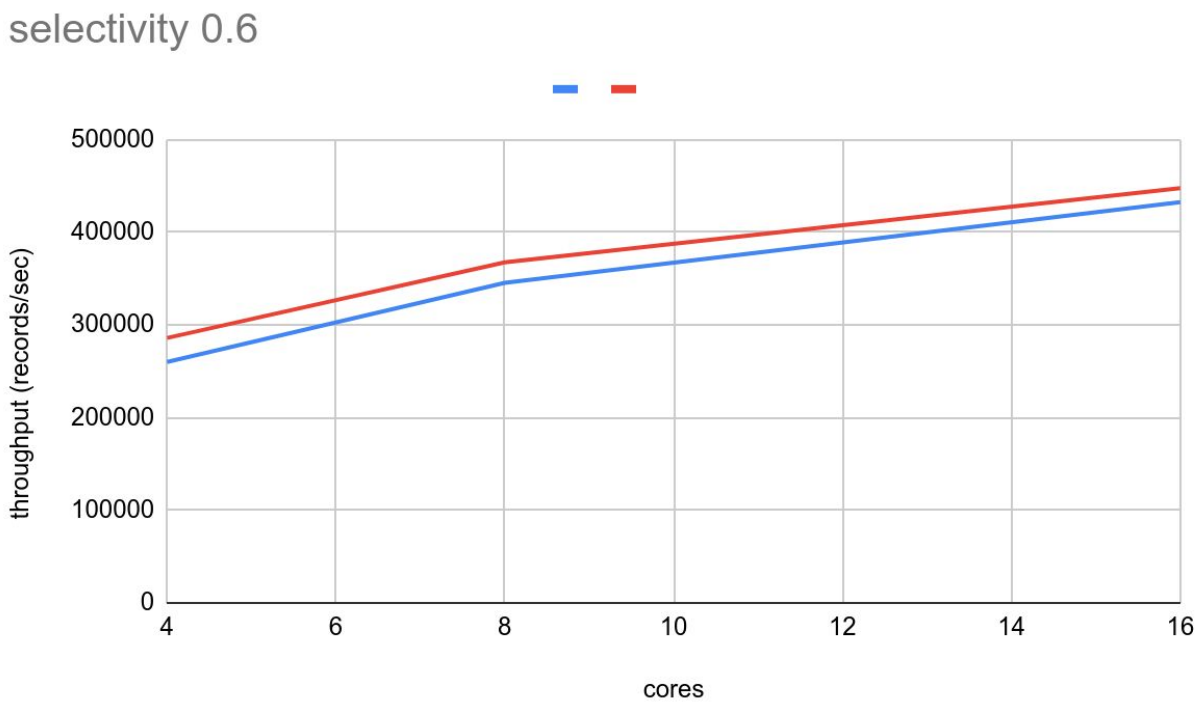


Fig. 50: Sustainable throughput for windowed join in both processing engines depending on the number of cores .

Σχολιασμός

Παρατηρούμε ότι το Flink είναι καλύτερο από το Spark . Αυτό οφείλεται σε δύο λόγους. Πρώτον το Spark έχει blocking operators.Αυτό σημαίνει ότι πρέπει να περιμένουμε να ολοκληρωθεί ένα στάδιο του operation για να πάμε στο επόμενο.Κάθε στάδιο ολοκληρώνεται όταν έχει επεξεργαστεί όλα τα στοιχεία που ανήκουν στο ίδιο window-batch. Όπως φαίνεται και στην παρακάτω εικόνα περιμένουμε να ολοκληρωθούν τα δύο active stages που κάνουν παράλληλα mapping στα records των δύο input streams για να πάμε στο pending stage του join.

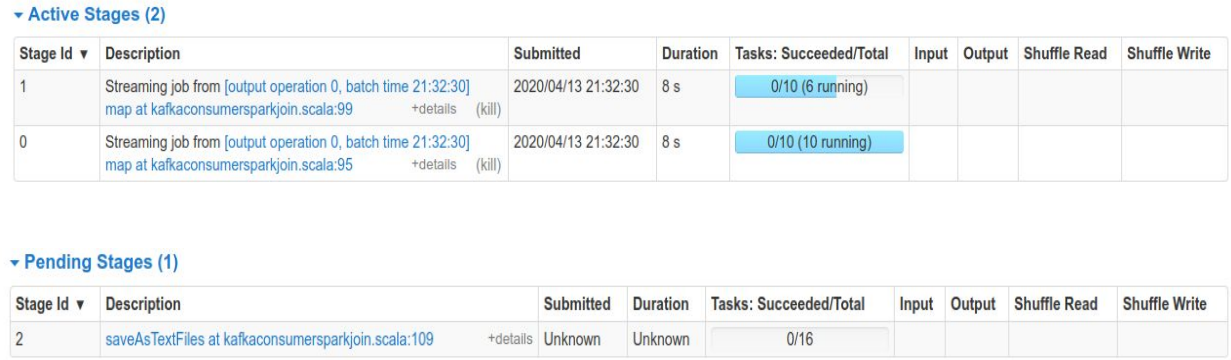


Fig. 51: Η σειρά εκτέλεσης των stages του window join στο Spark Streaming

Δεύτερον όπως και στο filter στο Spark παράγονται ενδιάμεσα Rdds σε κάθε στάδιο του DAG όπως φαίνεται και στον παρακάτω Spark graph για το join.

Fig. 52: DAG for join operation

Το Flink αντίθετα δεν έχει **blocking operators**. Αντιθέτως το Flink προωθεί κάθε record που έρχεται διαδοχικά σε όλα τα stages του operation υπό την προϋπόθεση ότι έχει ολοκληρωθεί η επεξεργασία του προηγούμενου record. (επεξεργασία ανά record). Τα στάδια ενός job λειτουργούν σαν αλυσίδα όπου κάθε στάδιο παίρνει επιτόπου records που έχει επεξεργαστεί το προηγούμενο στάδιο χωρίς να περιμένει να έχουν υποστεί επεξεργασία συγκεκριμένος αριθμός από records (batch).
Δεν παράγει ενδιάμεσα αποτελέσματα (όπως κάνει το Spark με τα ενδιάμεσα RDDs) , όπως φαίνεται και στο παρακάτω Flink graph .

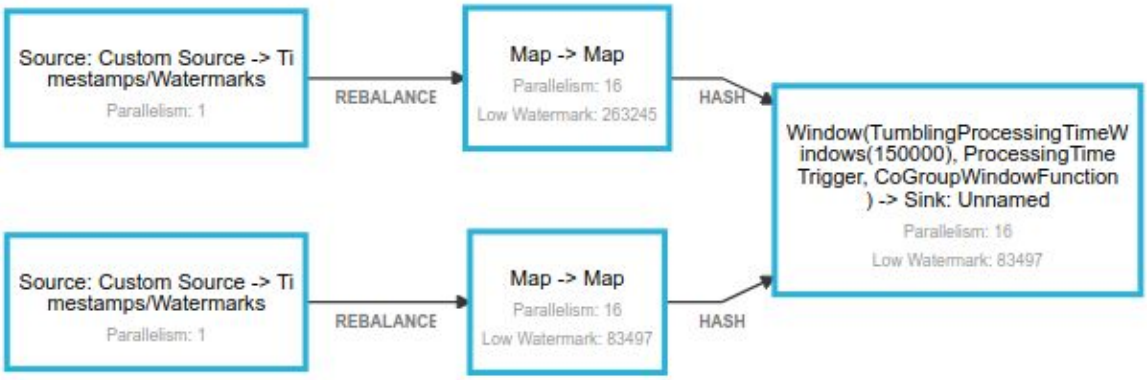


Fig. 53: Flink Dataflow for window operation

9.7.1.2 Selectivity

3ο πείραμα

Σε αυτό το πείραμα έχουμε ένα σταθερό setup στον cluster. Ο cluster αποτελείται από 5 nodes και 16 cores.

Μεταβάλλουμε το selectivity του συνολικού inputstream (αποτελείται από δύο διαφορετικά input streams) . Το πρώτο συνολικό input stream έχει selectivity 0.2 , το δεύτερο 0.6 και το τρίτο 0.8 .

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 1000 ms για όλα τις μετρήσεις του Spark . Το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **selectivity 0.2** το linger.ms για το application του Flink είναι 1000 ms **,**το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 30 . Για το application του Spark το batch size είναι 9.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 80 .

Στα **selectivity 0.6** το linger.ms για το application του Flink είναι 15.000 ms **,**το batch size είναι 250.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 80 . Για το application του Spark το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 20 .

Στο **selectivity 0.8** το linger.ms για το application του Flink είναι 1000 ms **,**το batch size είναι 12.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 90. Για το application του Spark το batch size είναι 8.000.000 bytes και ο αριθμός των partitions του καθενός από τα δύο input streams είναι 20 .

Latency

Στο **selectivity 0.2** το latency για το application του Flink είναι 1,24 .Για το application του Spark το latency είναι 1,16.

Στα **selectivity 0.6** το latency για το application του Flink είναι 1,32 .Για το application του Spark το latency είναι 1,24.

Στα **selectivity 0.8** το latency για το application του Flink είναι 1,43 .Για το application του Spark το latency είναι 1,20.

Το **διάγραμμα** απεικονίζει το μέσο maximum sustainable throughput σε συνάρτηση με τη μεταβολή του selectivity του inputstream για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

Κόκκινο: Flink
Μπλε : Spark

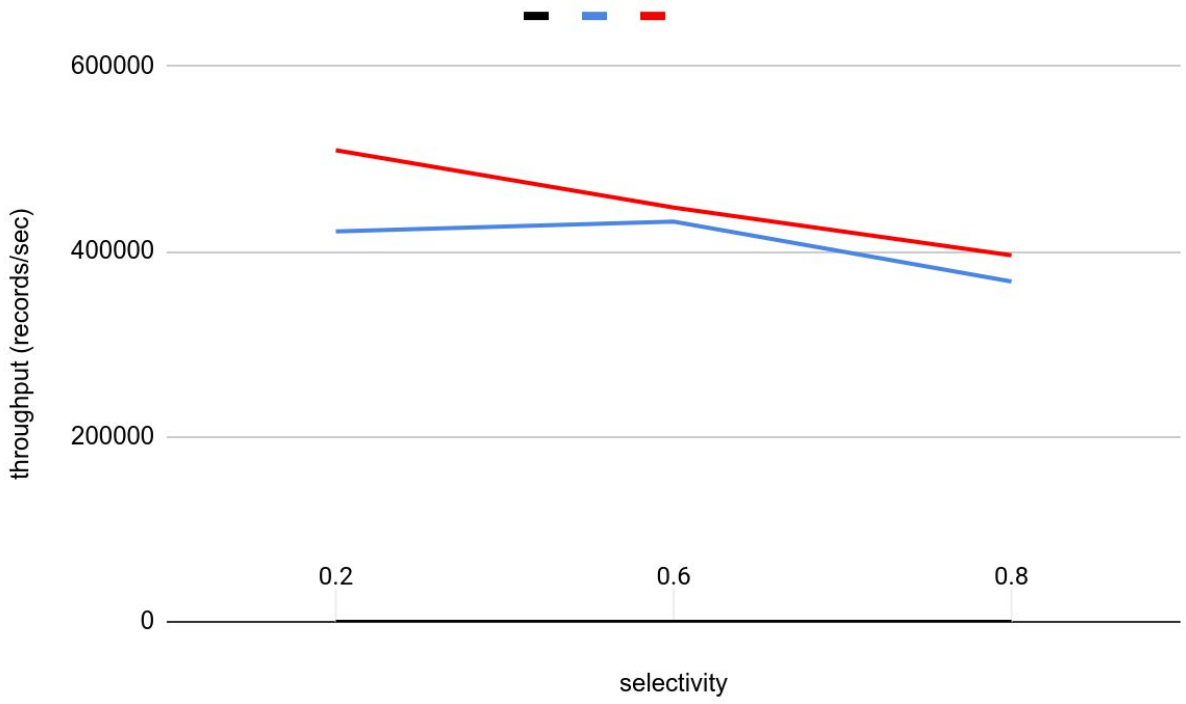


Fig. 54: Sustainable throughput for windowed join in both processing engines depending on the selectivity of the input streams .

Σχολιασμός

Παρατηρούμε ότι το Flink είναι καλύτερο από το Spark .Αυτό οφείλεται στο γεγονός ότι το Flink δεν έχει blocking operators και δεν παράγει ενδιάμεσα αποτελέσματα όπως προαναφέραμε.

Στο selectivity 0.2 είναι πολύ μεγάλη η διαφορά στο performance μεταξύ Spark και Flink . Στο selectivity 0.2 έχει μειωθεί κατά πολύ το output. Όσο μειώνεται το selectivity μειώνεται και το data shuffling (επιλέγονται λιγότερα στοιχεία των δύο streams) που πρέπει να κάνει το Flink στο στάδιο του join προκειμένου να φέρει στοιχεία με κοινό key και έτσι μεγαλώνει η διαφορά στο performance μεταξύ των δύο engines.

9.7.2 Δεύτερη περίπτωση

Join , tumbling window 2minutes,one dimension key

Για το join σε κάθε πείραμα μεταβάλλεται μια παράμετρος. Τα διαγράμματα είναι χωρισμένα με βάση την παράμετρο μεταβολής.

Στα **διαγράμματα** που ακολουθούν συγκρίνουμε το Spark και το Flink για κάθε πείραμα που διεξάγουμε σε σχέση με το **μέσο sustainable throughput** που αναφέραμε σε προηγούμενο κεφάλαιο.Το throughput μετριέται σε records/sec .

9.7.2.1 Cores

Για τον operator του join η μεταβολή του cluster έχει ως εξής : 4 nodes-4 cores
4 nodes-8 cores
5 nodes-16 cores

1ο πείραμα

Σε αυτό το πείραμα βάζουμε 2 input streams . Το **input stream1** αποτελείται από 15.170.000 tuples. Το **input stream 2** αποτελείται από 18.700.000 tuples.
Το **output** του join operation είναι 93.225.000 tuples και έχουμε selectivity 0.8.

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 5000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το batch size για το application του Flink είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 70 . Για το application του Spark το batch size είναι 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30 .

Στα **8 cores** το batch size για το application του Flink είναι 9.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 100. Για το application του Spark 9.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30 .

Στα **4 cores** το batch size για το application του Flink είναι 9.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30. Για το application του Spark 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 100.

Latency

Στα **16 cores** το latency για το application του Flink είναι 1,43 .Για το application του Spark το latency είναι 1,20 .

Στα **8 cores** το latency για το application του Flink είναι 2,11 .Για το application του Spark το latency είναι 2 .

Στα **4 cores** το latency για το application του Flink είναι 2,40 .Για το application του Spark το latency είναι 2,30.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink

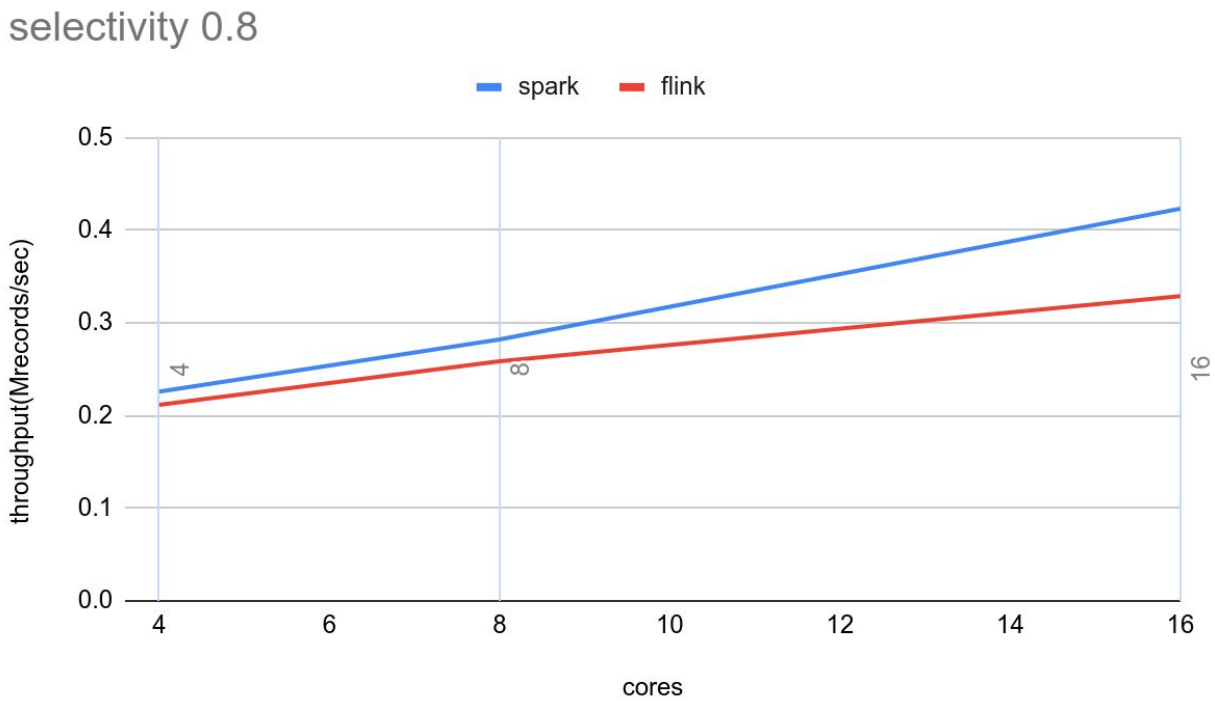


Fig. 55: Sustainable throughput for windowed join in both processing engines depending on the number of cores .

2ο πείραμα

Σε αυτό το πείραμα βάζουμε 2 input streams . Το **input stream1** αποτελείται από 15.170.000 tuples. Το **input stream 2** αποτελείται από 18.700.000 tuples.
Το **output** του join operation είναι 69.856.875 tuples και έχουμε selectivity 0.6.

Μεταβάλλουμε τα cores (και τα nodes) του cluster.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 5000 ms στα 16 και 8 cores. Στα 4 cores έχει τιμή 6000ms. Το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στα **16 cores** το batch size για το application του Flink είναι 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 60. Για το application του Spark το batch size είναι 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 10.

Στα **8 cores** το batch size για το application του Flink είναι 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 80. Για το application του Spark το batch size 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 20.

Στα **4 cores** το batch size για το application του Flink είναι 15.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 70. Για το application του Spark το batch size είναι 15.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 60.

Latency

Στα **16 cores** το latency για το application του Flink είναι 1,32 .Για το application του Spark το latency είναι 1,24 .

Στα **8 cores** το latency για το application του Flink είναι 1,51 .Για το application του Spark το latency είναι 1,41 .

Στα **4 cores** το latency για το application του Flink είναι 2,04 .Για το application του Spark το latency είναι 2,08.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή των cores για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του inputstream όπως έχουμε αναφέρει στο 6.11.

Κόκκινο: Flink
Μπλε : Spark

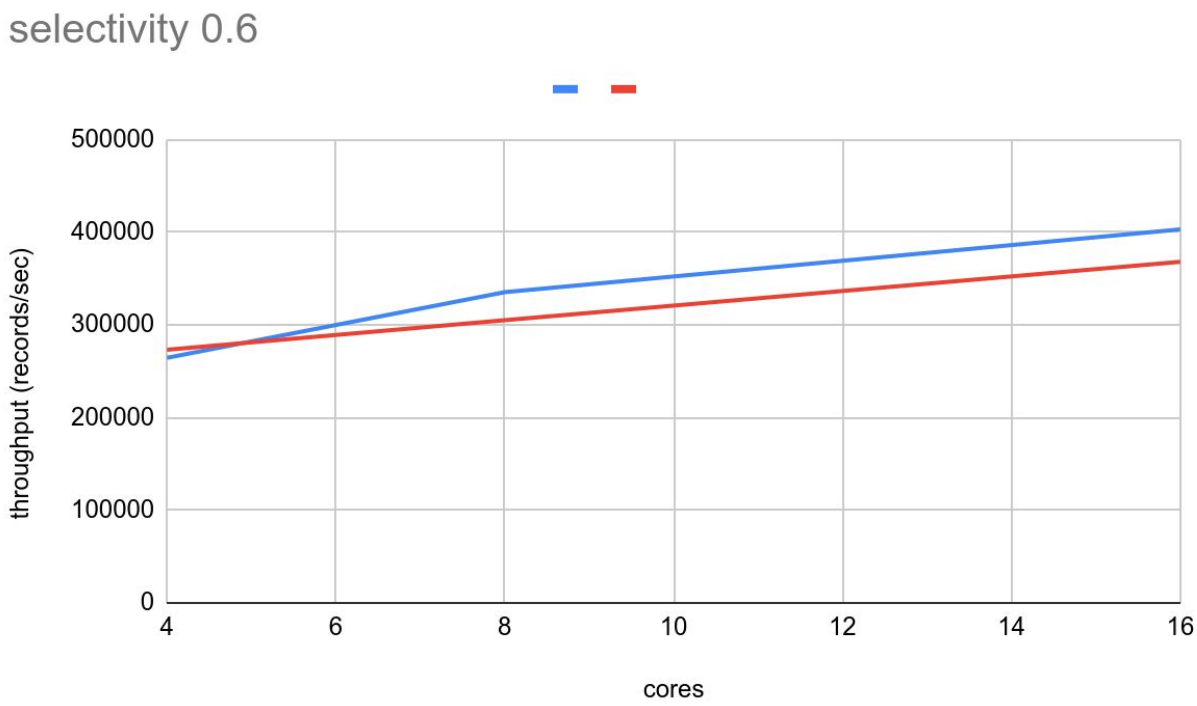


Fig. 56: Sustainable throughput for windowed join in both processing engines depending on the number of cores .

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink και στα δύο πειράματα.

Περιγραφή Flink graph

Στο Flink τα δεδομένα αφού κατανέμονται στα partitions των nodes στην συνέχεια υφίστανται ανακατανομή στα partitions των nodes με βάση το κλειδί (hash) προτού γίνει το join (αφού έχουμε keyed operation) . Άρα **το data shuffling είναι εκτεταμένο**.Στη συνέχεια πολλά parallel join tasks επικοινωνούν μεταξύ τους καθώς κάθε task πρέπει να έχει records των δύο input streams που να έχουν μεταξύ τους κοινά κλειδιά για να κάνει το join. Άρα **το communication είναι εκτεταμένο**.

Σε αυτό το use-case ,**το data shuffling και το communication** στο Flink είναι εκτεταμένο λόγω του γεγονότος ότι το key έχει **ένα dimension**. Για δεδομένο μέγεθος των input streams, όσο μικρότερο είναι το dimension του key των records τόσο αυξάνεται ο αριθμός των records των δύο streams που έχουν μεταξύ τους κοινό key. Αυτό σημαίνει ότι αυξάνεται το μέγεθος του join operation και συνεπώς αυξάνεται το data shuffling .
Συνεπώς το Flink χάνει χρόνο καθώς πολλά παράλληλα join tasks ανταλλάσουν records προκειμένου κάθε task να πάρει records των δύο input streams που να έχουν μεταξύ τους κοινά κλειδιά.

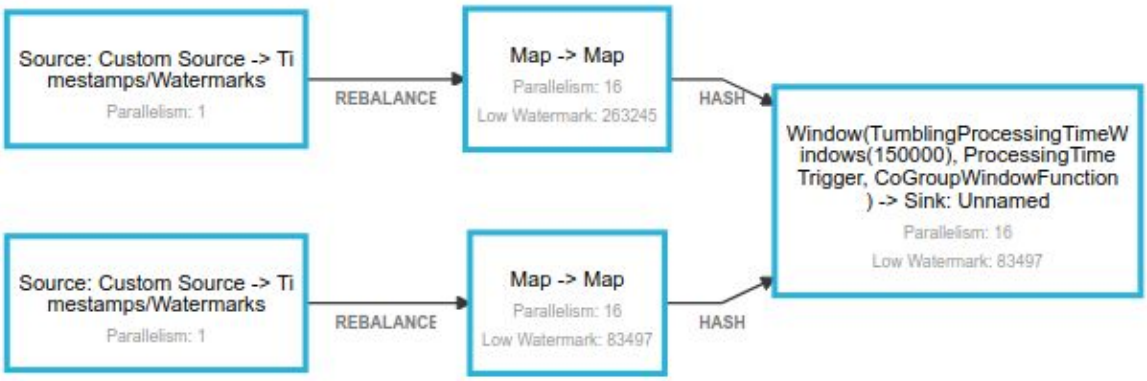


Fig. 57: Flink Dataflow for window join operation

Tree aggregate-tree reduce

Το Spark υλοποιεί ένα tree aggregate tree reduce pattern το οποίο **ακολουθεί το mapreduce μοντέλο και ελαχιστοποιεί το communication και το data shuffling.**
Αρχικά όλα τα εισερχόμενα στοιχεία κατανέμονται στους executors . Στη συνέχεια οι executors εκτελούν map operation όπου ανταλλάσουν μεταξύ τους **κάποια** στοιχεία προκειμένου όλα τα στοιχεία κάθε executor να ομαδοποιηθούν σε buckets ανάλογα με το κλειδί. Με αυτό τον τρόπο **ελαχιστοποιείται το data shuffling** καθώς έχουμε ανταλλαγή λίγων στοιχείων μεταξύ των executors.

Στη συνέχεια κάθε join task (που λειτουργεί ως reducer) καλεί εσωτερικά το co group και κάνει fetch buckets των δύο streams από τους executors που έχουν κοινά κλειδιά . Τα parallel join tasks στο Spark δεν επικοινωνούν μεταξύ τους καθώς το κάθε task έχει τα σωστά στοιχεία. Έτσι **ελαχιστοποιείται το communication**.

(Δεν τα πηγαίνει στα partitions για να κάνει join και μετά διαπιστώνει ότι δεν έχει records των δύο streams με κοινά keys και τα φέρνει από άλλα partitions όπως κάνει το Flink . Κατευθείαν παίρνει τα σωστά μέσω του co group πριν πάει στο στάδιο του join)

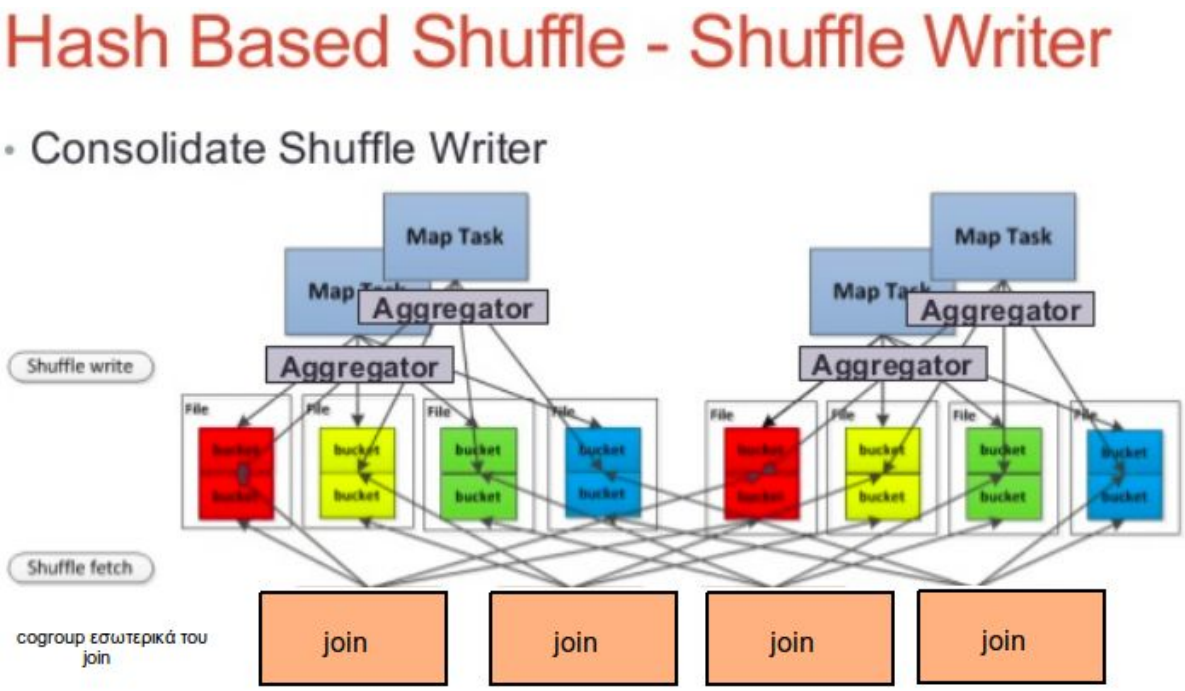


Fig. 58: Tree reduce, Tree join communication pattern στο Spark Streaming

Λόγοι για τη διαφορά στο performance

Ο λόγος για την διαφορά στο performance μεταξύ του Spark και του Flink βρίσκεται στον τρόπο με τον οποίο το Flink πραγματοποιεί aggregations και joins .Το Flink χρησιμοποιεί ένα task slot για κάθε operator instance. Όταν τα data είναι skewed (που σημαίνει ότι χρειάζονται ανακατανομή) το Flink σε πολλά παράλληλα instances του join (κάθε instance υλοποιείται σε

ένα slot) υλοποιεί ανακατανομή δεδομένων προκειμένου να έρθουν τα σωστά (στοιχεία των δύο streams που έχουν κοινά keys) και να γίνει το join.

Το Spark από την άλλη υλοποιεί tree aggregate -tree reduce αλγόριθμο ο οποίος μειώνει κατά πολύ τον όγκο των δεδομένων που πρέπει να μεταφερθούν μέσω του δικτύου

Τέλος στο selectivity 0.8 στο setup των 5 nodes (και 16 cores),η διαφορά μεταξύ του Spark και του Flink είναι πολύ μεγάλη. Αυτό οφείλεται στο limitation του network bandwidth. Όσο αυξάνονται τα nodes στο Flink το network bandwidth μπορεί να είναι ένα bottleneck . Στο selectivity 0.8 φαίνεται περισσότερο το limitation του network bandwidth γιατί όσο μεγαλύτερο είναι το selectivity τόσο αυξάνεται το data shuffling (αφού αυξάνεται το μέγεθος του join operation). Επειδή αυξάνεται το data shuffling χρησιμοποιείται περισσότερο το δίκτυο των nodes του Flink για να μεταφερθούν τα δεδομένα, χρειαζόμαστε περισσότερο network bandwidth και άρα έτσι γίνεται πιο αισθητό αυτό το limitation. Άρα το limitation του network bandwidth φαίνεται σε μεγαλύτερα selectivities.

9.7.2.2 Selectivity

3ο πείραμα

Σε αυτό το πείραμα έχουμε ένα σταθερό setup στον cluster. Ο cluster αποτελείται από 5 nodes και 16 cores.

Μεταβάλλουμε το **selectivity** του συνολικού inputstream (αποτελείται από δύο διαφορετικά input streams) . Το πρώτο συνολικό input stream έχει selectivity 0.2 , το δεύτερο 0.6 και το τρίτο 0.8 .

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 5000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στο **selectivity 0.2** το batch size για το application του Flink είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 70. Για το application του Spark το batch size είναι 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 60.

Στο **selectivity 0.6** το batch size για το application του Flink είναι 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 60 . Για το application του Spark 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 10.

Στο **selectivity 0.8** το batch size για το application του Flink είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 70 . Για το application του Spark 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30.

Latency

Στα **selectivity 0.2** το latency για το application του Flink είναι 1,24 .Για το application του Spark το latency είναι 1,04.

Στα **selectivity 0.6** το latency για το application του Flink είναι 1,32 .Για το application του Spark το latency είναι 1,16 .

Στα **selectivity 0.8** το latency για το application του Flink είναι 1,43 .Για το application του Spark το latency είναι 1,20.

Το **διάγραμμα** απεικονίζει το μέσο maximum sustainable throughput σε συνάρτηση με τη μεταβολή του selectivity του συνολικού inputstream για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του συνολικού inputstream όπως έχουμε αναφέρει στο 6.11.

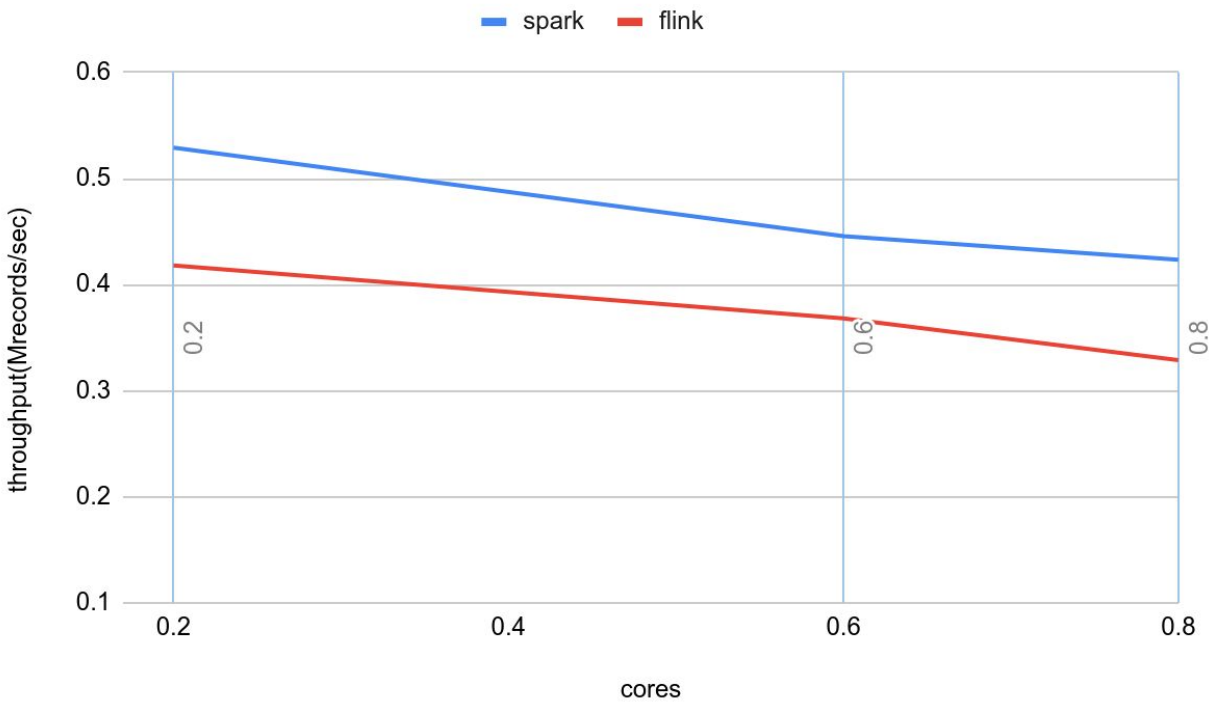


Fig. 59: Sustainable throughput for windowed join in both processing engines depending on the selectivity of the input streams .

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink . Αυτό οφείλεται στο γεγονός ότι το κλειδί έχει μια διάσταση , άρα αυξάνεται το μέγεθος του join operation σε κάθε selectivity και άρα το data shuffling. Αυτό σε συνδυασμό με το ότι το Spark χρησιμοποιεί το tree aggregate -tree reduce pattern που ελαχιστοποιεί τα δεδομένα που μεταφέρονται μέσω του δικτύου καθιστά το Spark καλύτερο από το Flink.

4ο πείραμα

Σε αυτό το πείραμα έχουμε ένα σταθερό setup στον cluster. Ο cluster αποτελείται από 5 nodes και 16 cores.

Μεταβάλλουμε το **selectivity** του συνολικού inputstream (αποτελείται από δύο διαφορετικά input streams) . Το πρώτο συνολικό input stream έχει selectivity 0.1 , το δεύτερο 0.5 και το τρίτο 0.8 . Τα δύο input streams σε κάθε selectivity έχουν **random κατανομή**.

Τιμές για τις παραμέτρους του Kafka producer

Το **linger.ms** έχει τιμή 5000 ms για όλα τις μετρήσεις και το **batch size** παίρνει διάφορες τιμές ανάλογα με τον αριθμό των nodes .Επίσης διαφοροποιείται και από το αν το application είναι Spark η Flink .

Στο **selectivity 0.1** το batch size για το application του Flink είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 20. Για το application του Spark το batch size είναι 12.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30.

Στο **selectivity 0.5** το batch size για το application του Flink είναι 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30 . Για το application του Spark 11.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 30.

Στο **selectivity 0.8** το batch size για το application του Flink είναι 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 50 . Για το application του Spark 10.000.000 bytes και ο αριθμός των partitions του καθένα από τα δύο input streams είναι 80.

Latency

Στα **selectivity 0.1** το latency για το application του Flink είναι 1,20 .Για το application του Spark το latency είναι 1,09.

Στα **selectivity 0.5** το latency για το application του Flink είναι 1,30 .Για το application του Spark το latency είναι 1,10.

Στα **selectivity 0.8** το latency για το application του Flink είναι 1,47 .Για το application του Spark το latency είναι 1,24.

Το **διάγραμμα** απεικονίζει το μέσο sustainable throughput σε συνάρτηση με τη μεταβολή του selectivity του συνολικού inputstream για το Spark και το Flink που έχει προκύψει από τα παραπάνω latencies και το αριθμό των στοιχείων του συνολικού inputstream όπως έχουμε αναφέρει στο 6.11.

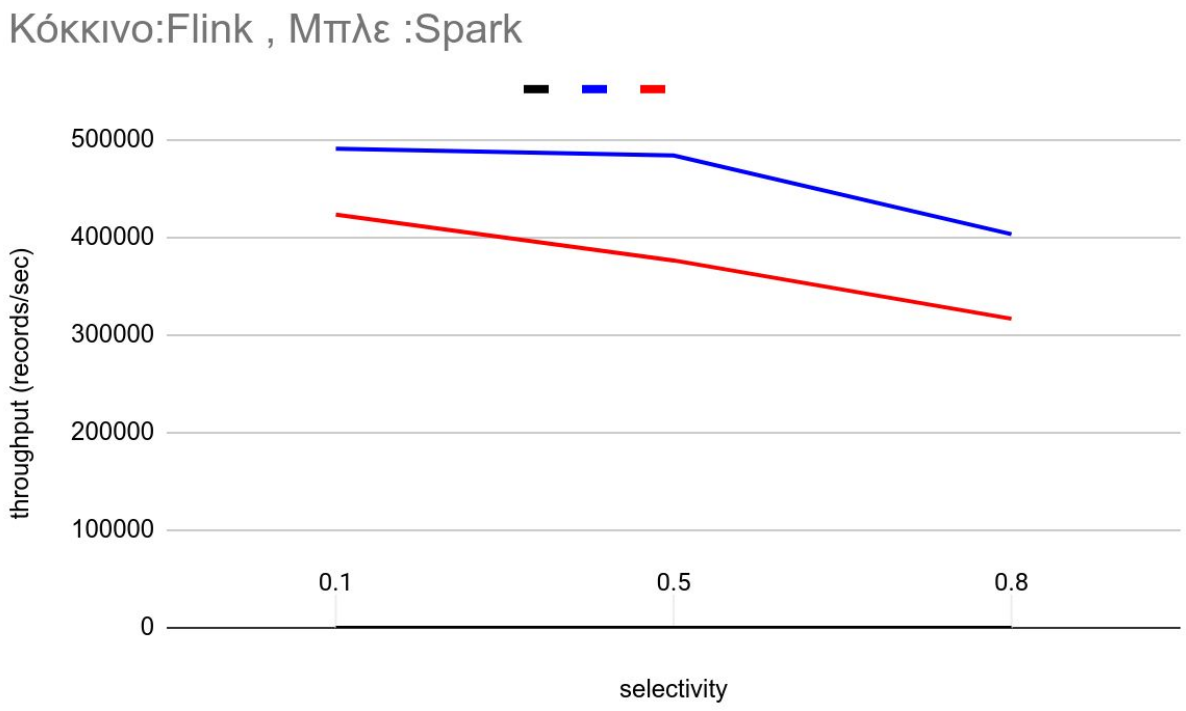


Fig. 60: Sustainable throughput for windowed join in both processing engines depending on the selectivity of the input streams .

Σχολιασμός

Παρατηρούμε ότι το Spark είναι καλύτερο από το Flink

Ιδανικά θέλουμε σε κάθε partition του Flink να έχουμε πολλά στοιχεία από τα δύο streams που να έχουν μεταξύ τους κοινά κλειδιά προκειμένου να μην τα ζητάμε από άλλα partitions (στο στάδιο του join). Στην περίπτωση μας κάτι τέτοιο δεν συμβαίνει καθώς οι κατανομές των δύο input streams σε κάθε selectivity, είναι τυχαίες και έτσι η πιθανότητα σε ένα partition να βρούμε πολλά στοιχεία από τα δύο streams που να έχουν μεταξύ τους κοινά κλειδιά είναι μικρή .Επίσης το key έχει **ένα dimension**. Για δεδομένο μέγεθος των input streams, όσο μικρότερο είναι το dimension του key των records τόσο αυξάνεται ο αριθμός των records των δύο streams που έχουν μεταξύ τους κοινό key. Αυτό σημαίνει ότι αυξάνεται το μέγεθος του join operation και συνεπώς αυξάνεται το data shuffling . Το Flink στο στάδιο του join χάνει πολύ χρόνο λόγω του εκτεταμένου data shuffling προκειμένου κάθε parallel join task να πάρει από άλλα partitions records των δύο streams που να έχουν μεταξύ τους κοινά κλειδιά. Γι αυτό το λόγο το Flink έχει χειρότερο performance από το Spark το οποίο χρησιμοποιεί το tree reduce και tree aggregate communication pattern που έχουμε ήδη περιγράψει και ελαχιστοποιεί το data shuffling και το communication.

ΚΕΦΑΛΑΙΟ 10: Συμπεράσματα

Πίνακας καταγραφής των βασικών διαφορών μεταξύ των δύο engines

	Spark Streaming	**Flink**
Notions of time	Processing time	Processing time, event time, ingestion time
timestamps	Όχι	ναι
backpressure	Ενεργοποιείται από το χρήστη	By default enabled
fault-tolerance	By default enabled. Τα εισερχόμενα data αντιγράφονται σε πολλαπλούς executors	Checkpointing. Ορίζεται από το χρήστη.
operators	blocking	non-blocking
Unit	Batch	record
Type of processing	Micro-batch processing	Real-time processing
windows	no triggers	triggers

TABLE III: Διαφορές μεταξύ Spark Streaming και Flink

Το Spark Streaming σε γενικές γραμμές έχει καλύτερο performance αναφορικά με το throughput σε σύγκριση με το Flink .Συγκεκριμένα αν τα inputstreams έχουν μιας διάστασης κλειδί το Spark έχει καλυτερο performance από το Flink **(Πείραμα aggregate by key, Πείραμα window join one-dimension key).** Είναι πιο επιρρεπές σε backpressure όπως διαπιστώσαμε και από τις μετρήσεις για την εύρεση του sustainable throughput καθώς εμφανίζει μεγαλύτερα fluctuations στο latency των operations όταν μεταβάλλεται ο αριθμός των kafka partitions στα input stream.

Το Flink έχει καλύτερο performance σε μεγάλα selectivities σε non-keyed operations **(Πείραμα filter)** .Επίσης έχει καλύτερο performance όταν τα inputstreams έχουν κλειδιά δύο διαστάσεων **(Πείραμα window join two-dimensional key).**
Εμείς σε αυτή τη μελέτη επικεντρωθήκαμε στην σύγκριση των δύο engines σε σχέση με το throughput. Σε σχέση με το latency που είναι μια εξίσου σημαντική μετρική , είναι πιθανό το Flink να είναι καλύτερο από το Spark από τη στιγμή που το Flink κάνει real-time processing και η επεξεργασία των δεδομένων γίνεται ανά στοιχείο.

ΚΕΦΑΛΑΙΟ 11: Βιβλιογραφία

[1] https://www.edureka.co/blog/spark-architecture/
[2] https://data-flair.training/blogs/limitations-of-apache-spark/
[3] https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/general_arch.html
[4] https://flink.apache.org/flink-architecture.html
[5] https://kafka.apache.org/intro
[6]https://www.cloudkarafka.com/blog/2016-11-30-part1-kafka-for-beginners-what-is-apache-kafka.html
[7] https://techvidvan.com/tutorials/apache-spark-dag-directed-acyclic-graph/
[8]https://medium.com/@rinu.gour123/kafka-performance-tuning-ways-for-kafka-optimization-fdee5b19505b
[9] https://ci.apache.org/projects/flink/flink-docs-release-1.7/concepts/programming-model.html
[10]https://hazelcast.com/glossary/micro-batch-processing/
[11]https://www.ververica.com/blog/how-flink-handles-backpressure
[12]https://de.slideshare.net/colorant/spark-shuffle-introduction
[13]https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/streaming/event_timestamps_watermarks.html
[14]http://vishnuviswanath.com/flink_trigger_evictor.html
[15]https://medium.com/swlh/exploit-apache-kafkas-message-format-to-save-storage-and-bandwidth-7e0c533edf26
[16]https://spark.apache.org/docs/latest/streaming-programming-guide.html
[17]https://www.tutorialspoint.com/apache_flink/apache_flink_architecture.htm
[18]https://medium.com/@patidarshailna/top-10-kafka-features-reasons-behind-the-popularity-of-apache-kafka-611d71877894
[19]https://hazelcast.com/glossary/stream-processing/
[20]https://www.ververica.com/what-is-stream-processing
[21]https://www.upsolver.com/blog/batch-stream-a-cheat-sheet
[22]https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.5/developing-storm-applications/content/understanding_sliding_and_tumbling_windows.html
[23]https://cwiki.apache.org/confluence/display/FLINK/Time+and+Order+in+Streams
[24]Jeyhun Karimov, Tilmann Rabl , Asterios Katsifodimos ,Roman Samarev , Henri Heiskanen and Volker Markl, “Benchmarking Distributed Stream Data Processing Systems”, 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1507-1518, IEEE, 2018